Compilers and Compiler Generators

an introduction with C++

© P.D. Terry, Rhodes University, 1996

p.terry@ru.ac.za

This is a set of Adobe PDF® files of the text of my book "Compilers and Compiler Generators - an introduction with C++", published in 1997 by International Thomson Computer Press. The original edition is now out of print, and the copyright has reverted to me.

The book is also available in other formats. The latest versions of the distribution and details of how to download up-to-date compressed versions of the text and its supporting software and courseware can be found at http://www.scifac.ru.ac.za/compilers/

The text of the book is Copyright © PD Terry. Although you are free to make use of the material for academic purposes, the material may not be redistributed without my knowledge or permission.

File List

- The 18 chapters of the book are filed as chap01.pdf through chap18.pdf
- The 4 appendices to the book are filed as appa.pdf through appd.pdf
- The original appendix A of the book is filed as appa0.pdf
- The contents of the book is filed as contents.pdf
- The preface of the book is filed as preface.pdf
- An index for the book is filed as index.pdf. Currently (January 2000) the page numbers refer to an A4 version in PCL® format available at http://www.scifac.ru.ac.za/compilers/longpcl.zip. However, software tools like GhostView may be used to search the files for specific text.
- The bibliography for the book is filed as biblio.pdf

Change List

- 18-October-1999 - Pre-release
- 12-November-1999 - First official on-line release
- 16-January-2000 - First release of Postscript version (incorporates minor corrections to chapter 12)
- 17-January-2000 - First release of PDF version

PREFACE

This book has been written to support a practically oriented course in programming language translation for senior undergraduates in Computer Science. More specifically, it is aimed at students who are probably quite competent in the art of imperative programming (for example, in C++, Pascal, or Modula-2), but whose mathematics may be a little weak; students who require only a solid introduction to the subject, so as to provide them with insight into areas of language design and implementation, rather than a deluge of theory which they will probably never use again; students who will enjoy fairly extensive case studies of translators for the sorts of languages with which they are most familiar; students who need to be made aware of compiler writing tools, and to come to appreciate and know how to use them. It will hopefully also appeal to a certain class of hobbyist who wishes to know more about how translators work.

The reader is expected to have a good knowledge of programming in an imperative language and, preferably, a knowledge of data structures. The book is practically oriented, and the reader who cannot read and write code will have difficulty following quite a lot of the discussion. However, it is difficult to imagine that students taking courses in compiler construction will not have that sort of background!

There are several excellent books already extant in this field. What is intended to distinguish this one from the others is that it attempts to mix theory and practice in a disciplined way, introducing the use of attribute grammars and compiler writing tools, at the same time giving a highly practical and pragmatic development of translators of only moderate size, yet large enough to provide considerable challenge in the many exercises that are suggested.

Overview

The book starts with a fairly simple overview of the translation process, of the constituent parts of a compiler, and of the concepts of porting and bootstrapping compilers. This is followed by a chapter on machine architecture and machine emulation, as later case studies make extensive use of code generation for emulated machines, a very common strategy in introductory courses. The next chapter introduces the student to the notions of regular expressions, grammars, BNF and EBNF, and the value of being able to specify languages concisely and accurately.

Two chapters follow that discuss simple features of assembler language, accompanied by the development of an assembler/interpreter system which allows not only for very simple assembly, but also for conditional assembly, macro-assembly, error detection, and so on. Complete code for such an assembler is presented in a highly modularized form, but with deliberate scope left for extensions, ranging from the trivial to the extensive.

Three chapters follow on formal syntax theory, parsing, and the manual construction of scanners and parsers. The usual classifications of grammars and restrictions on practical grammars are discussed in some detail. The material on parsing is kept to a fairly simple level, but with a thorough discussion of the necessary conditions for LL(1) parsing. The parsing method treated in most detail is the method of recursive descent, as is found in many Pascal compilers; LR parsing is only briefly discussed.

The next chapter is on syntax directed translation, and stresses to the reader the importance and usefulness of being able to start from a context-free grammar, adding attributes and actions that allow for the manual or mechanical construction of a program that will handle the system that it defines. Obvious applications come from the field of translators, but applications in other areas such as simple database design are also used and suggested.

The next two chapters give a thorough introduction to the use of Coco/R, a compiler generator based on L- attributed grammars. Besides a discussion of Cocol, the specification language for this tool, several in-depth case studies are presented, and the reader is given some indication of how parser generators are themselves constructed.

The next two chapters discuss the construction of a recursive descent compiler for a simple Pascal-like source language, using both hand-crafted and machine-generated techniques. The compiler produces pseudo-code for a hypothetical stack-based computer (for which an interpreter was developed in an earlier chapter). "On the fly" code generation is discussed, as well as the use of intermediate tree construction.

The last chapters extend the simple language (and its compiler) to allow for procedures and functions, demonstrate the usual stack-frame approach to storage management, and go on to discuss the implementation of simple concurrent programming. At all times the student can see how these are handled by the compiler/interpreter system, which slowly grows in complexity and usefulness until the final product enables the development of quite sophisticated programs.

The text abounds with suggestions for further exploration, and includes references to more advanced texts where these can be followed up. Wherever it seems appropriate the opportunity is taken to make the reader more aware of the strong and weak points in topical imperative languages. Examples are drawn from several languages, such as Pascal, Modula-2, Oberon, C, C++, Edison and Ada.

Support software

An earlier version of this text, published by Addison-Wesley in 1986, used Pascal throughout as a development tool. By that stage Modula-2 had emerged as a language far better suited to serious programming. A number of discerning teachers and programmers adopted it enthusiastically, and the material in the present book was originally and successfully developed in Modula-2. More recently, and especially in the USA, one has witnessed the spectacular rise in popularity of C++, and so as to reflect this trend, this has been adopted as the main language used in the present text. Although offering much of value to skilled practitioners, C++ is a complex language. As the aim of the text is not to focus on intricate C++programming, but compiler construction, the supporting software has been written to be as clear and as simple as possible. Besides the C++ code, complete source for all the case studies has also been provided on an accompanying IBM-PC compatible diskette in Turbo Pascal and Modula-2, so that readers who are proficient programmers in those languages but only have a reading knowledge of C++ should be able to use the material very successfully.

Appendix A gives instructions for unpacking the software provided on the diskette and installing it on a reader’s computer. In the same appendix will be found the addresses of various sites on the Internet where this software (and other freely available compiler construction software) can be found in various formats. The software provided on the diskette includes

- Emulators for the two virtual machines described in Chapter 4 (one of these is a simple accumulator based machine, the other is a simple stack based machine).
- The one- and two-pass assemblers for the accumulator based machine, discussed in Chapter 6.
- A macro assembler for the accumulator-based machine, discussed in Chapter 7.
- Three executable versions of the Coco/R compiler generator used in the text and described in detail in Chapter 12, along with the frame files that it needs. (The three versions produce Turbo Pascal, Modula-2 or C/C++ compilers)
- Complete source code for hand-crafted versions of each of the versions of the Clang compiler that is developed in a layered way in Chapters 14 through 18. This highly modularized code comes with an "on the fly" code generator, and also with an alternative code generator that builds and then walks a tree representation of the intermediate code.
- Cocol grammars and support modules for the numerous case studies throughout the book that use Coco/R. These include grammars for each of the versions of the Clang compiler.
- A program for investigating the construction of minimal perfect hash functions (as discussed in Chapter 14).
- A simple demonstration of an LR parser (as discussed in Chapter 10).

Use as a course text

The book can be used for courses of various lengths. By choosing a selection of topics it could be used on courses as short as 5-6 weeks (say 15-20 hours of lectures and 6 lab sessions). It could also be used to support longer and more intensive courses. In our university, selected parts of the material have been successfully used for several years in a course of about 35 - 40 hours of lectures with strictly controlled and structured, related laboratory work, given to students in a pre-Honours year. During that time the course has evolved significantly, from one in which theory and formal specification played a very low key, to the present stage where students have come to appreciate the use of specification and syntax-directed compiler-writing systems as very powerful and useful tools in their armoury.

It is hoped that instructors can select material from the text so as to suit courses tailored to their own interests, and to their students' capabilities. The core of the theoretical material is to be found in Chapters 1, 2, 5, 8, 9, 10 and 11, and it is suggested that this material should form part of any course based on the book. Restricting the selection of material to those chapters would deny the student the very important opportunity to see the material in practice, and at least a partial selection of the material in the practically oriented chapters should be studied. However, that part of the material in Chapter 4 on the accumulator-based machine, and Chapters 6 and 7 on writing assemblers for this machine could be omitted without any loss of continuity. The development of the small Clang compiler in Chapters 14 through 18 is handled in a way that allows for the later sections of Chapter 15, and for Chapters 16 through 18 to be omitted if time is short. A very wide variety of laboratory exercises can be selected from those suggested as exercises, providing the students with both a challenge, and a feeling of satisfaction when they rise to meet that challenge. Several of these exercises are based on the idea of developing a small compiler for a language

similar to the one discussed in detail in the text. Development of such a compiler could rely entirely on traditional hand-crafted techniques, or could rely entirely on a tool-based approach (both approaches have been successfully used at our university). If a hand-crafted approach were used, Chapters 12 and 13 could be omitted; Chapter 12 is largely a reference manual in any event, and could be left to the students to study for themselves as the need arose. Similarly, Chapter 3 falls into the category of background reading.

At our university we have also used an extended version of the Clang compiler as developed in the text (one incorporating several of the extensions suggested as exercises) as a system for students to study concurrent programming *per se*, and although it is a little limited, it is more than adequate for the purpose. We have also used a slightly extended version of the assembler program very successfully as our primary tool for introducing students to the craft of programming at the assembler level.

Limitations

It is, perhaps, worth a slight digression to point out some things which the book does not claim to be, and to justify some of the decisions made in the selection of material.

In the first place, while it is hoped that it will serve as a useful foundation for students who are already considerably more advanced, a primary aim has been to make the material as accessible as possible to students with a fairly limited background, to enhance the background, and to make them somewhat more critical of it. In many cases this background is still Pascal based; increasingly it is tending to become C++ based. Both of these languages have become rather large and complex, and I have found that many students have a very superficial idea of how they really fit together. After a course such as this one, many of the pieces of the language jigsaw fit together rather better.

When introducing the use of compiler writing tools, one might follow the many authors who espouse the classic lex/yacc approach. However, there are now a number of excellent LL(1) based tools, and these have the advantage that the code which is produced is close to that which might be hand-crafted; at the same time, recursive descent parsing, besides being fairly intuitive, is powerful enough to handle very usable languages.

That the languages used in case studies and their translators are relative toys cannot be denied. The Clang language of later chapters, for example, supports only integer variables and simple one-dimensional arrays of these, and has concurrent features allowing little beyond the simulation of some simple textbook examples. The text is not intended to be a comprehensive treatise on systems programming in general, just on certain selected topics in that area, and so very little is said about native machine code generation and optimization, linkers and loaders, the interaction and relationship with an operating system, and so on. These decisions were all taken deliberately, to keep the material readily understandable and as machine-independent as possible. The systems may be toys, but they are very usable toys! Of course the book is then open to the criticism that many of the more difficult topics in translation (such as code generation and optimization) are effectively not covered at all, and that the student may be deluded into thinking that these areas do not exist. This is not entirely true; the careful reader will find most of these topics mentioned somewhere.

Good teachers will always want to put something of their own into a course, regardless of the quality of the prescribed textbook. I have found that a useful (though at times highly dangerous) technique is deliberately not to give the best solutions to a problem in a class discussion, with the

optimistic aim that students can be persuaded to "discover" them for themselves, and even gain a sense of achievement in so doing. When applied to a book the technique is particularly dangerous, but I have tried to exploit it on several occasions, even though it may give the impression that the author is ignorant.

Another dangerous strategy is to give too much away, especially in a book like this aimed at courses where, so far as I am aware, the traditional approach requires that students make far more of the design decisions for themselves than my approach seems to allow them. Many of the books in the field do not show enough of how something is actually done: the bridge between what they give and what the student is required to produce is in excess of what is reasonable for a course which is only part of a general curriculum. I have tried to compensate by suggesting what I hope is a very wide range of searching exercises. The solutions to some of these are well known, and available in the literature. Again, the decision to omit explicit references was deliberate (perhaps dangerously so). Teachers often have to find some way of persuading the students to search the literature for themselves, and this is not done by simply opening the journal at the right page for them.

Acknowledgements

I am conscious of my gratitude to many people for their help and inspiration while this book has been developed.

Like many others, I am grateful to Niklaus Wirth, whose programming languages and whose writings on the subject of compiler construction and language design refute the modern trend towards ever-increasing complexity in these areas, and serve as outstanding models of the way in which progress should be made.

This project could not have been completed without the help of Hanspeter Mössenböck (author of the original Coco/R compiler generator) and Francisco Arzu (who ported it to C++), who not only commented on parts of the text, but also willingly gave permission for their software to be distributed with the book. My thanks are similarly due to Richard Cichelli for granting permission to distribute (with the software for Chapter 14) a program based on one he wrote for computing minimal perfect hash functions, and to Christopher Cockburn for permission to include his description of tonic sol-fa (used in Chapter 13).

I am grateful to Volker Pohlers for help with the port of Coco/R to Turbo Pascal, and to Dave Gillespie for developing `p2c`, a most useful program for converting Modula-2 and Pascal code to C/C++.

I am deeply indebted to my colleagues Peter Clayton, George Wells and Peter Wentworth for many hours of discussion and fruitful suggestions. John Washbrook carefully reviewed the manuscript, and made many useful suggestions for its improvement. Shaun Bangay patiently provided incomparable technical support in the installation and maintenance of my hardware and software, and rescued me from more than one disaster when things went wrong. To Rhodes University I am indebted for the use of computer facilities, and for granting me leave to complete the writing of the book. And, of course, several generations of students have contributed in intangible ways by their reaction to my courses.

The development of the software in this book relied heavily on the use of electronic mail, and I am grateful to Randy Bush, compiler writer and network guru extraordinaire, for his friendship, and for his help in making the Internet a reality in developing countries in Africa and elsewhere.

But, as always, the greatest debt is owed to my wife Sally and my children David and Helen, for their love and support through the many hours when they must have wondered where my priorities lay.

Pat Terry
Rhodes University
Grahamstown

Trademarks

Ada is a trademark of the US Department of Defense.
Apple II is a trademark of Apple Corporation.
Borland C++, Turbo C++, TurboPascal and Delphi are trademarks of Borland International Corporation.
GNU C Compiler is a trademark of the Free Software Foundation.
IBM and IBM PC are trademarks of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
MC68000 and MC68020 are trademarks of Motorola Corporation.
MIPS is a trademark of MIPS computer systems.
Microsoft, MS and MS-DOS are registered trademarks and Windows is a trademark of Microsoft Corporation.
SPARC is a trademark of Sun Microsystems.
Stony Brook Software and QuickMod are trademarks of Gogesch Micro Systems, Inc.
occam and Transputer are trademarks of Inmos.
UCSD Pascal and UCSD p-System are trademarks of the Regents of the University of California.
UNIX is a registered trademark of AT&T Bell Laboratories.
Z80 is a trademark of Zilog Corporation.

COMPILERS AND COMPILER GENERATORS

an introduction with C++

© P.D. Terry, Rhodes University, 1996

e-mail p.terry@ru.ac.za

The Postscript ® edition of this book was derived from the on-line versions available at http://www.scifac.ru.ac.za/compilers/, a WWW site that is occasionally updated, and which contains the latest versions of the various editions of the book, with details of how to download compressed versions of the text and its supporting software and courseware.

The original edition of this book, published originally by International Thomson, is now out of print, but has a home page at http://cs.ru.ac.za/homes/cspt/compbook.htm. In preparing the on-line edition, the opportunity was taken to correct the few typographical mistakes that crept into the first printing, and to create a few hyperlinks to where the source files can be found.

Feel free to read and use this book for study or teaching, but please respect my copyright and do not distribute it further without my consent. If you do make use of it I would appreciate hearing from you.

CONTENTS

Preface

Acknowledgements

1 Introduction

1.1 Objectives
1.2 Systems programs and translators
1.3 The relationship between high-level languages and translators

2 Translator classification and structure

2.1 T-diagrams
2.2 Classes of translator
2.3 Phases in translation
2.4 Multi-stage translators
2.5 Interpreters, interpretive compilers, and emulators

3 Compiler construction and bootstrapping

3.1 Using a high-level host language
3.2 Porting a high-level translator

3.3 Bootstrapping
3.4 Self-compiling compilers
3.5 The half bootstrap
3.6 Bootstrapping from a portable interpretive compiler
3.7 A P-code assembler

4 Machine emulation

4.1 Simple machine architecture
4.2 Addressing modes
4.3 Case study 1 - a single-accumulator machine
4.4 Case study 2 - a stack-oriented computer

5 Language specification

5.1 Syntax, semantics, and pragmatics
5.2 Languages, symbols, alphabets and strings
5.3 Regular expressions
5.4 Grammars and productions
5.5 Classic BNF notation for productions
5.6 Simple examples
5.7 Phrase structure and lexical structure
5.8 ε-productions
5.9 Extensions to BNF
5.10 Syntax diagrams
5.11 Formal treatment of semantics

6 Simple assemblers

6.1 A simple ASSEMBLER language
6.2 One- and two-pass assemblers, and symbol tables
6.3 Towards the construction of an assembler
6.4 Two-pass assembly
6.5 One-pass assembly

7 Advanced assembler features

7.1 Error detection
7.2 Simple expressions as addresses
7.3 Improved symbol table handling - hash tables
7.4 Macro-processing facilities
7.5 Conditional assembly
7.6 Relocatable code
7.7 Further projects

8 Grammars and their classification

8.1 Equivalent grammars
8.2 Case study - equivalent grammars for describing expressions
8.3 Some simple restrictions on grammars

8.4 Ambiguous grammars
8.5 Context sensitivity
8.6 The Chomsky hierarchy
8.7 Case study - Clang

9 Deterministic top-down parsing

9.1 Deterministic top-down parsing
9.2 Restrictions on grammars so as to allow LL(1) parsing
9.3 The effect of the LL(1) conditions on language design

10 Parser and scanner construction

10.1 Construction of simple recursive descent parsers
10.2 Case studies
10.3 Syntax error detection and recovery
10.4 Construction of simple scanners
10.5 Case studies
10.6 LR parsing
10.7 Automated construction of scanners and parsers

11 Syntax-directed translation

11.1 Embedding semantic actions into syntax rules
11.2 Attribute grammars
11.3 Synthesized and inherited attributes
11.4 Classes of attribute grammars
11.5 Case study - a small student database

12 Using Coco/R - overview

12.1 Installing and running Coco/R
12.2 Case study - a simple adding machine
12.3 Scanner specification
12.4 Parser specification
12.5 The driver program

13 Using Coco/R - Case studies

13.1 Case study - Understanding C declarations
13.2 Case study - Generating one-address code from expressions
13.3 Case study - Generating one-address code from an AST
13.4 Case study - How do parser generators work?
13.5 Project suggestions

14 A simple compiler - the front end

14.1 Overall compiler structure
14.2 Source handling
14.3 Error reporting

14.4 Lexical analysis
14.5 Syntax analysis
14.6 Error handling and constraint analysis
14.7 The symbol table handler
14.8 Other aspects of symbol table management - further types

15 A simple compiler - the back end

15.1 The code generation interface
15.2 Code generation for a simple stack machine
15.3 Other aspects of code generation

16 Simple block structure

16.1 Parameterless procedures
16.2 Storage management

17 Parameters and functions

17.1 Syntax and semantics
17.2 Symbol table support for context sensitive features
17.3 Actual parameters and stack frames
17.4 Hypothetical stack machine support for parameter passing
17.5 Context sensitivity and LL(1) conflict resolution
17.6 Semantic analysis and code generation
17.7 Language design issues

18 Concurrent programming

18.1 Fundamental concepts
18.2 Parallel processes, exclusion and synchronization
18.3 A semaphore-based system - syntax, semantics, and code generation
18.4 Run-time implementation

Appendix A: Software resources for this book

Appendix B: Source code for the Clang compiler/interpreter

Appendix C: Cocol grammar for the Clang compiler/interpreter

Appendix D: Source code for a macro assembler

Bibliography

Index

1 INTRODUCTION

1.1 Objectives

The use of computer languages is an essential link in the chain between human and computer. In this text we hope to make the reader more aware of some aspects of

- Imperative programming languages - their syntactic and semantic features; the ways of specifying syntax and semantics; problem areas and ambiguities; the power and usefulness of various features of a language.

- Translators for programming languages - the various classes of translator (assemblers, compilers, interpreters); implementation of translators.

- Compiler generators - tools that are available to help automate the construction of translators for programming languages.

This book is a complete revision of an earlier one published by Addison-Wesley (Terry, 1986). It has been written so as not to be too theoretical, but to relate easily to languages which the reader already knows or can readily understand, like Pascal, Modula-2, C or C++. The reader is expected to have a good background in one of those languages, access to a good implementation of it, and, preferably, some background in assembly language programming and simple machine architecture. We shall rely quite heavily on this background, especially on the understanding the reader should have of the meaning of various programming constructs.

Significant parts of the text concern themselves with case studies of actual translators for simple languages. Other important parts of the text are to be found in the many exercises and suggestions for further study and experimentation on the part of the reader. In short, the emphasis is on "doing" rather than just "reading", and the reader who does not attempt the exercises will miss many, if not most, of the finer points.

The primary language used in the implementation of our case studies is C++ (Stroustrup, 1990). Machine readable source code for all these case studies is to be found on the IBM-PC compatible diskette that is included with the book. As well as C++ versions of this code, we have provided equivalent source in Modula-2 and Turbo Pascal, two other languages that are eminently suitable for use in a course of this nature. Indeed, for clarity, some of the discussion is presented in a pseudo-code that often resembles Modula-2 rather more than it does C++. It is only fair to warn the reader that the code extracts in the book are often just that - extracts - and that there are many instances where identifiers are used whose meaning may not be immediately apparent from their local context. The conscientious reader will have to expend some effort in browsing the code. Complete source for an assembler and interpreter appears in the appendices, but the discussion often revolves around simplified versions of these programs that are found in their entirety only on the diskette.

1.2 Systems programs and translators

Users of modern computing systems can be divided into two broad categories. There are those who never develop their own programs, but simply use ones developed by others. Then there are those who are concerned as much with the development of programs as with their subsequent use. This latter group - of whom we as computer scientists form a part - is fortunate in that program development is usually aided by the use of high-level languages for expressing algorithms, the use of interactive editors for program entry and modification, and the use of sophisticated job control languages or graphical user interfaces for control of execution. Programmers armed with such tools have a very different picture of computer systems from those who are presented with the hardware alone, since the use of compilers, editors and operating systems - a class of tools known generally as **systems programs** - removes from humans the burden of developing their systems at the machine level. That is not to claim that the use of such tools removes all burdens, or all possibilities for error, as the reader will be well aware.

Well within living memory, much program development was done in machine language - indeed, some of it, of necessity, still is - and perhaps some readers have even tried this for themselves when experimenting with microprocessors. Just a brief exposure to programs written as almost meaningless collections of binary or hexadecimal digits is usually enough to make one grateful for the presence of high-level languages, clumsy and irritating though some of their features may be.

However, in order for high-level languages to be usable, one must be able to convert programs written in them into the binary or hexadecimal digits and bitstrings that a machine will understand. At an early stage it was realized that if constraints were put on the syntax of a high-level language the translation process became one that could be automated. This led to the development of **translators** or **compilers** - programs which accept (as data) a textual representation of an algorithm expressed in a **source language**, and which produce (as primary output) a representation of the same algorithm expressed in another language, the **object** or **target language**.

Beginners often fail to distinguish between the compilation (*compile-time*) and execution (*run-time*) phases in developing and using programs written in high-level languages. This is an easy trap to fall into, since the translation (compilation) is often hidden from sight, or invoked with a special function key from within an integrated development environment that may possess many other magic function keys. Furthermore, beginners are often taught programming with this distinction deliberately blurred, their teachers offering explanations such as "when a computer executes a *read* statement it reads a number from the input data into a variable". This hides several low-level operations from the beginner. The underlying implications of file handling, character conversion, and storage allocation are glibly ignored - as indeed is the necessity for the computer to be programmed to understand the word *read* in the first place. Anyone who has attempted to program input/output (I/O) operations directly in assembler languages will know that many of them are non-trivial to implement.

A translator, being a program in its own right, must itself be written in a computer language, known as its **host** or **implementation language**. Today it is rare to find translators that have been developed from scratch in machine language. Clearly the first translators had to be written in this way, and at the outset of translator development for any new system one has to come to terms with the machine language and machine architecture for that system. Even so, translators for new machines are now invariably developed in high-level languages, often using the techniques of **cross-compilation** and **bootstrapping** that will be discussed in more detail later.

The first major translators written may well have been the Fortran compilers developed by Backus

and his colleagues at IBM in the 1950's, although machine code development aids were in existence by then. The first Fortran compiler is estimated to have taken about 18 person-years of effort. It is interesting to note that one of the primary concerns of the team was to develop a system that could produce object code whose efficiency of execution would compare favourably with that which expert human machine coders could achieve. An automatic translation process can rarely produce code as optimal as can be written by a really skilled user of machine language, and to this day important components of systems are often developed at (or very near to) machine level, in the interests of saving time or space.

Translator programs themselves are never completely portable (although parts of them may be), and they usually depend to some extent on other systems programs that the user has at his or her disposal. In particular, input/output and file management on modern computer systems are usually controlled by the **operating system**. This is a program or suite of programs and routines whose job it is to control the execution of other programs so as best to share resources such as printers, plotters, disk files and tapes, often making use of sophisticated techniques such as parallel processing, multiprogramming and so on. For many years the development of operating systems required the use of programming languages that remained closer to the machine code level than did languages suitable for scientific or commercial programming. More recently a number of successful higher level languages have been developed with the express purpose of catering for the design of operating systems and real-time control. The most obvious example of such a language is C, developed originally for the implementation of the UNIX operating system, and now widely used in all areas of computing.

1.3 The relationship between high-level languages and translators

The reader will rapidly become aware that the design and implementation of translators is a subject that may be developed from many possible angles and approaches. The same is true for the design of programming languages.

Computer languages are generally classed as being "high-level" (like Pascal, Fortran, Ada, Modula-2, Oberon, C or C++) or "low-level" (like ASSEMBLER). High-level languages may further be classified as "imperative" (like all of those just mentioned), or "functional" (like Lisp, Scheme, ML, or Haskell), or "logic" (like Prolog).

High-level languages are claimed to possess several advantages over low-level ones:

- *Readability:* A good high-level language will allow programs to be written that in some ways resemble a quasi-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting, a highly desirable property when one considers that many programs are written once, but possibly studied by humans many times thereafter.

- *Portability:* High-level languages, being essentially machine independent, hold out the promise of being used to develop portable software. This is software that can, in principle (and even occasionally in practice), run unchanged on a variety of different machines - provided only that the source code is recompiled as it moves from machine to machine.

  To achieve machine independence, high-level languages may deny access to low-level features, and are sometimes spurned by programmers who have to develop low-level machine dependent systems. However, some languages, like C and Modula-2, were specifically designed to allow access to these features from within the context of high-level constructs.

- *Structure and object orientation:* There is general agreement that the structured programming movement of the 1960's and the object-oriented movement of the 1990's have resulted in a great improvement in the quality and reliability of code. High-level languages can be designed so as to encourage or even subtly enforce these programming paradigms.

- *Generality:* Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

- *Brevity:* Programs expressed in high-level languages are often considerably shorter (in terms of their number of source lines) than their low-level equivalents.

- *Error checking:* Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages - or at least their implementations - can, and often do, enforce a great deal of error checking both at compile-time and at run-time. For this they are, of course, often criticized by programmers who have to develop time-critical code, or who want their programs to abort as quickly as possible.

These advantages sometimes appear to be over-rated, or at any rate, hard to reconcile with reality. For example, readability is usually within the confines of a rather stilted style, and some beginners are disillusioned when they find just how unnatural a high-level language is. Similarly, the generality of many languages is confined to relatively narrow areas, and programmers are often dismayed when they find areas (like string handling in standard Pascal) which seem to be very poorly handled. The explanation is often to be found in the close coupling between the development of high-level languages and of their translators. When one examines successful languages, one finds numerous examples of compromise, dictated largely by the need to accommodate language ideas to rather uncompromising, if not unsuitable, machine architectures. To a lesser extent, compromise is also dictated by the quirks of the interface to established operating systems on machines. Finally, some appealing language features turn out to be either impossibly difficult to implement, or too expensive to justify in terms of the machine resources needed. It may not immediately be apparent that the design of Pascal (and of several of its successors such as Modula-2 and Oberon) was governed partly by a desire to make it easy to compile. It is a tribute to its designer that, in spite of the limitations which this desire naturally introduced, Pascal became so popular, the model for so many other languages and extensions, and encouraged the development of superfast compilers such as are found in Borland's Turbo Pascal and Delphi systems.

The design of a programming language requires a high degree of skill and judgement. There is evidence to show that one's language is not only useful for expressing one's ideas. Because language is also used to formulate and develop ideas, one's knowledge of language largely determines *how* and, indeed, *what* one can think. In the case of programming languages, there has been much controversy over this. For example, in languages like Fortran - for long the *lingua franca* of the scientific computing community - recursive algorithms were "difficult" to use (not impossible, just difficult!), with the result that many programmers brought up on Fortran found recursion strange and difficult, even something to be avoided at all costs. It is true that recursive algorithms are sometimes "inefficient", and that compilers for languages which allow recursion may exacerbate this; on the other hand it is also true that some algorithms are more simply explained in a recursive way than in one which depends on explicit repetition (the best examples probably being those associated with tree manipulation).

There are two divergent schools of thought as to how programming languages should be designed. The one, typified by the Wirth school, stresses that languages should be small and understandable,

and that much time should be spent in consideration of what tempting features might be omitted without crippling the language as a vehicle for system development. The other, beloved of languages designed by committees with the desire to please everyone, packs a language full of every conceivable potentially useful feature. Both schools claim success. The Wirth school has given us Pascal, Modula-2 and Oberon, all of which have had an enormous effect on the thinking of computer scientists. The other approach has given us Ada, C and C++, which are far more difficult to master well and extremely complicated to implement correctly, but which claim spectacular successes in the marketplace.

Other aspects of language design that contribute to success include the following:

- *Orthogonality:* Good languages tend to have a small number of well thought out features that can be combined in a logical way to supply more powerful building blocks. Ideally these features should not interfere with one another, and should not be hedged about by a host of inconsistencies, exceptional cases and arbitrary restrictions. Most languages have blemishes - for example, in Wirth's original Pascal a function could only return a scalar value, not one of any structured type. Many potentially attractive extensions to well-established languages prove to be extremely vulnerable to unfortunate oversights in this regard.

- *Familiar notation:* Most computers are "binary" in nature. Blessed with ten toes on which to check out their number-crunching programs, humans may be somewhat relieved that high-level languages usually make decimal arithmetic the rule, rather than the exception, and provide for mathematical operations in a notation consistent with standard mathematics. When new languages are proposed, these often take the form of derivatives or dialects of well-established ones, so that programmers can be tempted to migrate to the new language and still feel largely at home - this was the route taken in developing C++ from C, Java from C++, and Oberon from Modula-2, for example.

Besides meeting the ones mentioned above, a successful modern high-level language will have been designed to meet the following additional criteria:

- *Clearly defined:* It must be clearly described, for the benefit of both the user and the compiler writer.

- *Quickly translated:* It should admit quick translation, so that program development time when using the language is not excessive.

- *Modularity:* It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.

- *Efficient:* It should permit the generation of efficient object code.

- *Widely available:* It should be possible to provide translators for all the major machines and for all the major operating systems.

The importance of a clear language description or specification cannot be over-emphasized. This must apply, firstly, to the so-called **syntax** of the language - that is, it must specify accurately what form a source program may assume. It must apply, secondly, to the so-called **static semantics** of the language - for example, it must be clear what constraints must be placed on the use of entities of differing types, or the scope that various identifiers have across the program text. Finally, the

specification must also apply to the **dynamic semantics** of programs that satisfy the syntactic and static semantic rules - that is, it must be capable of predicting the effect any program expressed in that language will have when it is executed.

Programming language description is extremely difficult to do accurately, especially if it is attempted through the medium of potentially confusing languages like English. There is an increasing trend towards the use of formalism for this purpose, some of which will be illustrated in later chapters. Formal methods have the advantage of precision, since they make use of the clearly defined notations of mathematics. To offset this, they may be somewhat daunting to programmers weak in mathematics, and do not necessarily have the advantage of being very concise - for example, the informal description of Modula-2 (albeit slightly ambiguous in places) took only some 35 pages (Wirth, 1985), while a formal description prepared by an ISO committee runs to over 700 pages.

Formal specifications have the added advantage that, in principle, and to a growing degree in practice, they may be used to help automate the implementation of translators for the language. Indeed, it is increasingly rare to find modern compilers that have been implemented without the help of so-called **compiler generators**. These are programs that take a formal description of the syntax and semantics of a programming language as input, and produce major parts of a compiler for that language as output. We shall illustrate the use of compiler generators at appropriate points in our discussion, although we shall also show how compilers may be crafted by hand.

Exercises

1.1 Make a list of as many translators as you can think of that can be found on your computer system.

1.2 Make a list of as many other systems programs (and their functions) as you can think of that can be found on your computer system.

1.3 Make a list of existing features in your favourite (or least favourite) programming language that you find irksome. Make a similar list of features that you would like to have seen added. Then examine your lists and consider which of the features are probably related to the difficulty of implementation.

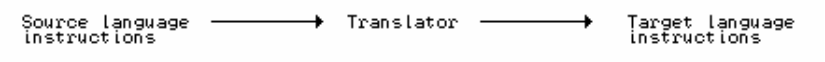
Further reading

As we proceed, we hope to make the reader more aware of some of the points raised in this section. Language design is a difficult area, and much has been, and continues to be, written on the topic. The reader might like to refer to the books by Tremblay and Sorenson (1985), Watson (1989), and Watt (1991) for readable summaries of the subject, and to the papers by Wirth (1974, 1976a, 1988a), Kernighan (1981), Welsh, Sneeringer and Hoare (1977), and Cailliau (1982). Interesting background on several well-known languages can be found in *ACM SIGPLAN Notices* for August 1978 and March 1993 (Lee and Sammet, 1978, 1993), two special issues of that journal devoted to the history of programming language development. Stroustrup (1993) gives a fascinating exposition of the development of C++, arguably the most widely used language at the present time. The terms "static semantics" and "dynamic semantics" are not used by all authors; for a discussion on this point see the paper by Meek (1990).

2 TRANSLATOR CLASSIFICATION AND STRUCTURE

In this chapter we provide the reader with an overview of the inner structure of translators, and some idea of how they are classified.

A translator may formally be defined as a function, whose domain is a source language, and whose range is contained in an object or target language.



A little experience with translators will reveal that it is rarely considered part of the translator's function to execute the algorithm expressed by the source, merely to change its representation from one form to another. In fact, at least three languages are involved in the development of translators: the source language to be translated, the object or target language to be generated, and the host language to be used for implementing the translator. If the translation takes place in several stages, there may even be other, intermediate, languages. Most of these - and, indeed, the host language and object languages themselves - usually remain hidden from a user of the source language.

2.1 T-diagrams

A useful notation for describing a computer program, particularly a translator, uses so-called **T-diagrams**, examples of which are shown in Figure 2.1.

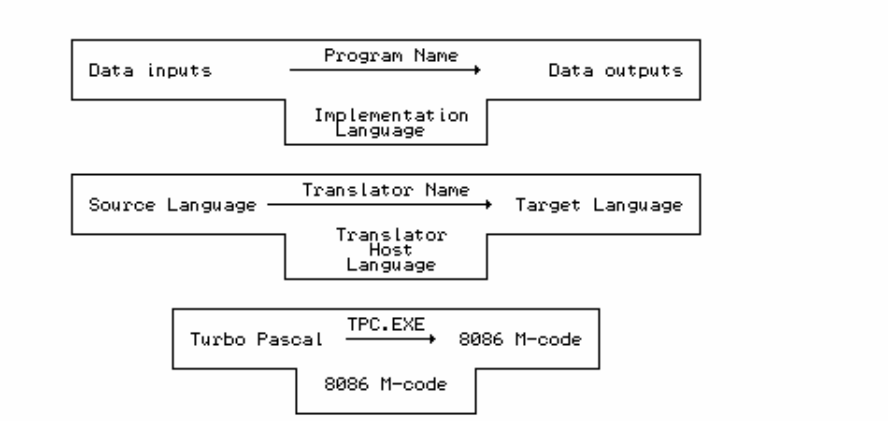


Figure 2.1 T-diagrams. (a) A general program (b) a general translator (c) A Turbo Pascal compiler for an MS-DOS system

We shall use the notation "M-code" to stand for "machine code" in these diagrams. Translation itself is represented by standing the T on a machine, and placing the source program and object program on the left and right arms, as depicted in Figure 2.2.

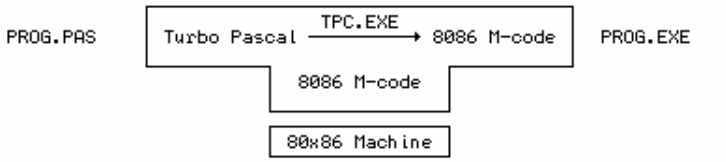


Figure 2.2 A Turbo Pascal compilation on an 80x86 machine

We can also regard this particular combination as depicting an **abstract machine** (sometimes called a **virtual machine**), whose aim in life is to convert Turbo Pascal source programs into their 8086 machine code equivalents.

T-diagrams were first introduced by Bratman (1961). They were further refined by Earley and Sturgis (1970), and are also used in the books by Bennett (1990), Watt (1993), and Aho, Sethi and Ullman (1986).

2.2 Classes of translator

It is common to distinguish between several well-established classes of translator:

- The term **assembler** is usually associated with those translators that map low-level language instructions into machine code which can then be executed directly. Individual source language statements usually map one-for-one to machine-level instructions.

- The term **macro-assembler** is also associated with those translators that map low-level language instructions into machine code, and is a variation on the above. Most source language statements map one- for-one into their target language equivalents, but some *macro* statements map into a sequence of machine- level instructions - effectively providing a text replacement facility, and thereby extending the assembly language to suit the user. (This is not to be confused with the use of procedures or other subprograms to "extend" high-level languages, because the method of implementation is usually very different.)

- The term **compiler** is usually associated with those translators that map high-level language instructions into machine code which can then be executed directly. Individual source language statements usually map into many machine-level instructions.

- The term **pre-processor** is usually associated with those translators that map a superset of a high-level language into the original high-level language, or that perform simple text substitutions before translation takes place. The best-known pre-processor is probably that which forms an integral part of implementations of the language C, and which provides many of the features that contribute to the widely- held perception that C is the only really portable language.

- The term **high-level translator** is often associated with those translators that map one high-level language into another high-level language - usually one for which sophisticated compilers already exist on a range of machines. Such translators are particularly useful as components of a two-stage compiling system, or in assisting with the bootstrapping techniques to be discussed shortly.

- The terms **decompiler** and **disassembler** refer to translators which attempt to take object code at a low level and regenerate source code at a higher level. While this can be done quite successfully for the production of assembler level code, it is much more difficult when one tries to recreate source code originally written in, say, Pascal.

Many translators generate code for their host machines. These are called **self-resident translators**. Others, known as **cross-translators**, generate code for machines other than the host machine. Cross-translators are often used in connection with microcomputers, especially in embedded systems, which may themselves be too small to allow self-resident translators to operate satisfactorily. Of course, cross-translation introduces additional problems in connection with transferring the object code from the donor machine to the machine that is to execute the translated program, and can lead to delays and frustration in program development.

The output of some translators is absolute machine code, left loaded at fixed locations in a machine ready for immediate execution. Other translators, known as **load-and-go** translators, may even initiate execution of this code. However, a great many translators do not produce fixed-address machine code. Rather, they produce something closely akin to it, known as **semicompiled** or **binary symbolic** or **relocatable** form. A frequent use for this is in the development of composite libraries of special purpose routines, possibly originating from a mixture of source languages. Routines compiled in this way are linked together by programs called **linkage editors** or **linkers**, which may be regarded almost as providing the final stage for a multi-stage translator. Languages that encourage the separate compilation of parts of a program - like Modula-2 and C++ - depend critically on the existence of such linkers, as the reader is doubtless aware. For developing really large software projects such systems are invaluable, although for the sort of "throw away" programs on which most students cut their teeth, they can initially appear to be a nuisance, because of the overheads of managing several files, and of the time taken to link their contents together.

T-diagrams can be combined to show the interdependence of translators, loaders and so on. For example, the FST Modula-2 system makes use of a compiler and linker as shown in Figure 2.3.

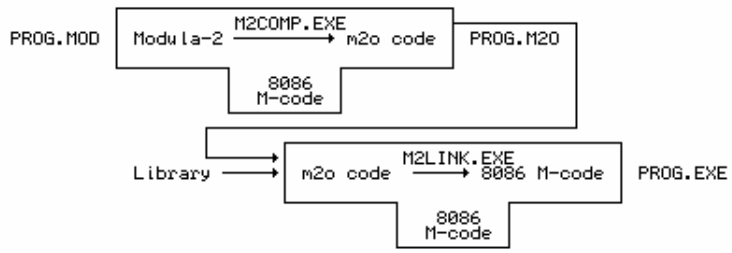


Figure 2.3 Compiling and linking Modula-2 program on the FST system

---

Exercises

2.1 Make a list of as many translators as you can think of that can be found on your system.

2.2 Which of the translators known to you are of the load-and-go type?

2.3 Do you know whether any of the translators you use produce relocatable code? Is this of a standard form? Do you know the names of the linkage editors or loaders used on your system?

2.3 Phases in translation

Translators are highly complex programs, and it is unreasonable to consider the translation process as occurring in a single step. It is usual to regard it as divided into a series of **phases**. The simplest breakdown recognizes that there is an **analytic phase**, in which the source program is analysed to determine whether it meets the syntactic and static semantic constraints imposed by the language. This is followed by a **synthetic phase** in which the corresponding object code is generated in the target language. The components of the translator that handle these two major phases are said to comprise the **front end** and the **back end** of the compiler. The front end is largely independent of the target machine, the back end depends very heavily on the target machine. Within this structure we can recognize smaller components or phases, as shown in Figure 2.4.

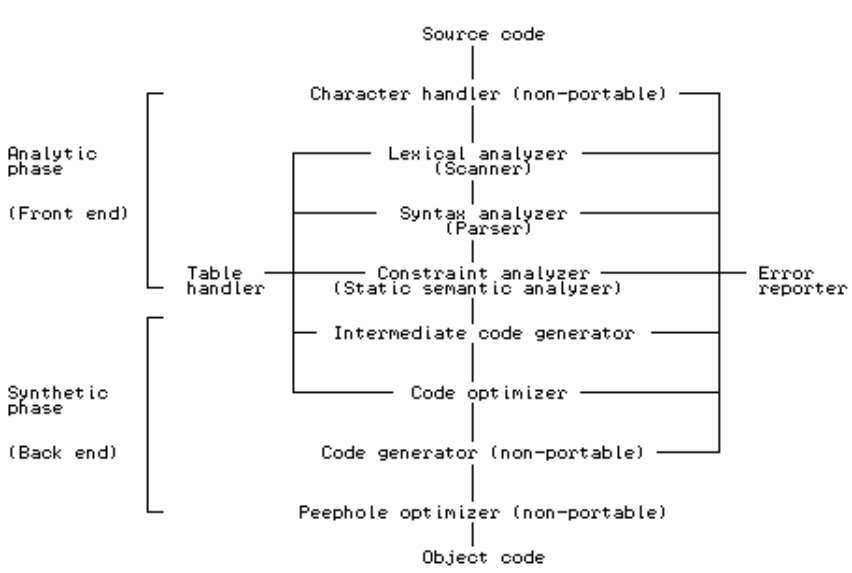


Figure 2.4 Structure and phases of a compiler

The **character handler** is the section that communicates with the outside world, through the operating system, to read in the characters that make up the source text. As character sets and file handling vary from system to system, this phase is often machine or operating system dependent.

The **lexical analyser** or **scanner** is the section that fuses characters of the source text into groups that logically make up the **tokens** of the language - symbols like identifiers, strings, numeric constants, keywords like `while` and `if`, operators like `<=`, and so on. Some of these symbols are very simply represented on the output from the scanner, some need to be associated with various properties such as their names or values.

Lexical analysis is sometimes easy, and at other times not. For example, the Modula-2 statement

```
WHILE A > 3 * B DO A := A - 1 END
```

easily decodes into tokens

```
WHILE       keyword
A           identifier              name A
```

```
>           operator                comparison
3           constant literal        value 3
*           operator                multiplication
B           identifier              name B
DO          keyword
A           identifier              name A
:=          operator                assignment
A           identifier              name A
-           operator                subtraction
1           constant literal        value 1
END         keyword
```

as we read it from left to right, but the Fortran statement

```
        10      DO 20 I = 1 . 30
```

is more deceptive. Readers familiar with Fortran might see it as decoding into

```
10          label
DO          keyword
20          statement label
I           INTEGER identifier
=           assignment operator
1           INTEGER constant literal
,           separator
30          INTEGER constant literal
```

while those who enjoy perversity might like to see it as it really is:

```
10          label
DO20I       REAL identifier
=           assignment operator
1.30        REAL constant literal
```

One has to look quite hard to distinguish the period from the "expected" comma. (Spaces are irrelevant in Fortran; one would, of course *be* perverse to use identifiers with unnecessary and highly suggestive spaces in them.) While languages like Pascal, Modula-2 and C++ have been cleverly designed so that lexical analysis can be clearly separated from the rest of the analysis, the same is obviously not true of Fortran and other languages that do not have reserved keywords.

The **syntax analyser** or **parser** groups the tokens produced by the scanner into syntactic structures - which it does by parsing expressions and statements. (This is analogous to a human analysing a sentence to find components like "subject", "object" and "dependent clauses"). Often the parser is combined with the **contextual constraint analyser**, whose job it is to determine that the components of the syntactic structures satisfy such things as scope rules and type rules within the context of the structure being analysed. For example, in Modula-2 the syntax of a *while* statement is sometimes described as

```
WHILE  Expression  DO  StatementSequence  END
```

It is reasonable to think of a statement in the above form with any type of *Expression* as being syntactically correct, but as being devoid of real meaning unless the value of the *Expression* is constrained (in this context) to be of the Boolean type. No program really has any meaning until it is executed dynamically. However, it is possible with strongly typed languages to predict at compile-time that some source programs can have no sensible meaning (that is, statically, before an attempt is made to execute the program dynamically). Semantics is a term used to describe "meaning", and so the constraint analyser is often called the **static semantic analyser**, or simply the semantic analyser.

The output of the syntax analyser and semantic analyser phases is sometimes expressed in the form of a decorated **abstract syntax tree** (AST). This is a very useful representation, as it can be used in clever ways to optimize code generation at a later stage.

Whereas the **concrete syntax** of many programming languages incorporates many keywords and tokens, the **abstract syntax** is rather simpler, retaining only those components of the language needed to capture the real content and (ultimately) meaning of the program. For example, whereas the concrete syntax of a *while* statement requires the presence of `WHILE`, `DO` and `END` as shown above, the essential components of the *while* statement are simply the (Boolean) *Expression* and the statements comprising the *StatementSequence*.

Thus the Modula-2 statement

```
WHILE  (1 < P)  AND  (P < 9)  DO  P := P + Q  END
```

or its C++ equivalent

```
while  (1 < P && P < 9)  P = P + Q;
```

are both depicted by the common AST shown in Figure 2.5.



Figure 2.5 AST for the statement while (1 < P && P < 9) P = P + Q;

An abstract syntax tree on its own is devoid of some semantic detail; the semantic analyser has the task of adding "type" and other contextual information to the various nodes (hence the term "decorated" tree).

Sometimes, as for example in the case of most Pascal compilers, the construction of such a tree is not explicit, but remains implicit in the recursive calls to procedures that perform the syntax and semantic analysis.

Of course, it is also possible to construct concrete syntax trees. The Modula-2 form of the statement

```
WHILE  (1 < P)  AND  (P < 9)  DO  P := P + Q  END
```

could be depicted in full and tedious detail by the tree shown in Figure 2.6. The reader may have to make reference to Modula-2 syntax diagrams and the knowledge of Modula-2 precedence rules to understand why the tree looks so complicated.

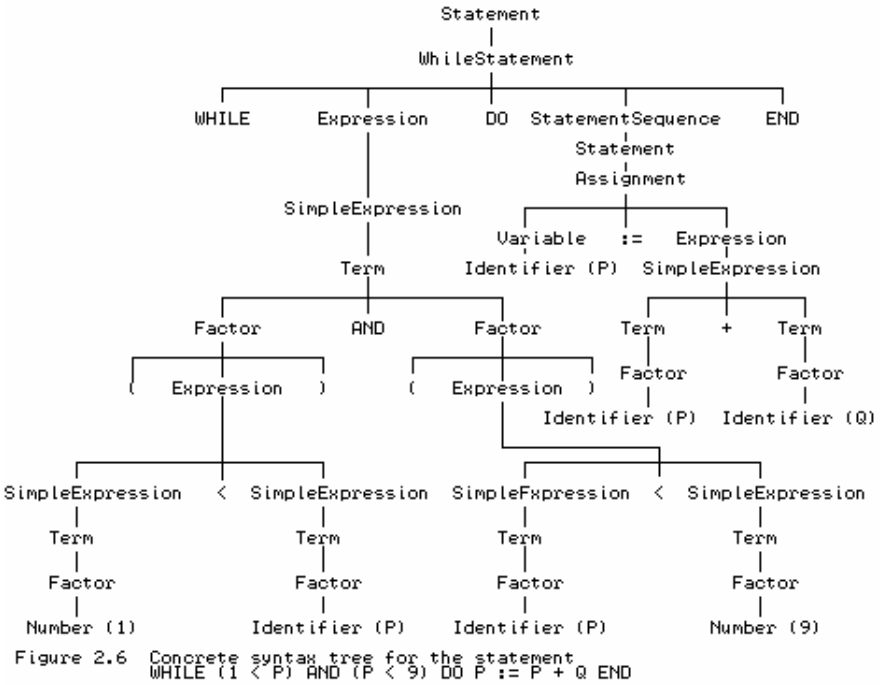


Figure 2.6 Concrete syntax tree for the statement
WHILE (1 < P) AND (P < 9) DO P := P + Q END

The phases just discussed are all analytic in nature. The ones that follow are more synthetic. The first of these might be an **intermediate code generator**, which, in practice, may also be integrated with earlier phases, or omitted altogether in the case of some very simple translators. It uses the data structures produced by the earlier phases to generate a form of code, perhaps in the form of simple code skeletons or macros, or ASSEMBLER or even high-level code for processing by an external assembler or separate compiler. The major difference between intermediate code and actual machine code is that intermediate code need not specify in detail such things as the exact machine registers to be used, the exact addresses to be referred to, and so on.

Our example statement

```
WHILE  (1 < P)  AND  (P < 9)  DO   P := P + Q  END
```

might produce intermediate code equivalent to

```
L0      if 1 < P goto L1
        goto L3
L1      if P < 9 goto L2
        goto L3
L2      P := P + Q
        goto L0
L3      continue
```

Then again, it might produce something like

```
L0      T1 := 1 < P
        T2 := P < 9
        if T1 and T2 goto L1
        goto L2
L1      P := P + Q
        goto L0
L2      continue
```

depending on whether the implementors of the translator use the so-called *sequential conjunction* or *short-circuit* approach to handling compound Boolean expressions (as in the first case) or the so-called *Boolean operator* approach. The reader will recall that Modula-2 and C++ require the short-circuit approach. However, the very similar language Pascal did not specify that one approach

be preferred above the other.

A **code optimizer** may optionally be provided, in an attempt to improve the intermediate code in the interests of speed or space or both. To use the same example as before, obvious optimization would lead to code equivalent to

```
L0      if 1 >= P goto L1
        if P >= 9 goto L1
        P := P + Q
        goto L0
L1      continue
```

The most important phase in the back end is the responsibility of the **code generator**. In a real compiler this phase takes the output from the previous phase and produces the object code, by deciding on the memory locations for data, generating code to access such locations, selecting registers for intermediate calculations and indexing, and so on. Clearly this is a phase which calls for much skill and attention to detail, if the finished product is to be at all efficient. Some translators go on to a further phase by incorporating a so-called **peephole optimizer** in which attempts are made to reduce unnecessary operations still further by examining short sequences of generated code in closer detail.

Below we list the actual code generated by various MS-DOS compilers for this statement. It is readily apparent that the code generation phases in these compilers are markedly different. Such differences can have a profound effect on program size and execution speed.

```
Borland C++ 3.1 (47 bytes)

CS:A0 BBB702     MOV  BX,02B7
CS:A3 C746FE5100 MOV  WORD PTR[BP-2],0051
CS:A8 EB07       JMP  B1
CS:AA 8BC3       MOV  AX,BX
CS:AC 0346FE     ADD  AX,[BP-2]
CS:AF 8BD8       MOV  BX,AX
CS:B1 83FB01     CMP  BX,1
CS:B4 7E05       JLE  BB
CS:B6 B80100     MOV  AX,1
CS:B9 EB02       JMP  BD
CS:BB 33C0       XOR  AX,AX
CS:BD 50         PUSH AX
CS:BE 83FB09     CMP  BX,9
CS:C1 7D05       JGE  C8
CS:C3 B80100     MOV  AX,1
CS:C6 EB02       JMP  CA
CS:C8 33C0       XOR  AX,AX
CS:CA 5A         POP  DX
CS:CB 85D0       TEST DX,AX
CS:CD 75DB       JNZ  AA
```

```
Turbo Pascal (46 bytes)
(with no short circuit evaluation)

CS:09 833E3E0009 CMP  WORD PTR[003E],9
CS:0E 7C04       JL   14
CS:10 B000       MOV  AL,0
CS:12 EB02       JMP  16
CS:14 B001       MOV  AL,1
CS:16 8AD0       MOV  DL,AL
CS:18 833E3E0001 CMP  WORD PTR[003E],1
CS:1D 7F04       JG   23
CS:1F B000       MOV  AL,0
CS:21 EB02       JMP  25
CS:23 B001       MOV  AL,01
CS:25 22C2       AND  AL,DL
CS:27 08C0       OR   AL,AL
CS:29 740C       JZ   37
CS:2B A13E00     MOV  AX,[003E]
CS:2E 03064000   ADD  AX,[0040]
CS:32 A33E00     MOV  [003E],AX
CS:35 EBD2       JMP  9
```

```
JPI TopSpeed Modula-2 (29 bytes)

CS:19 2E         CS:
CS:1A 8E1E2700   MOV  DS,[0027]
CS:1E 833E000001 CMP  WORD PTR[0000],1
CS:23 7E11       JLE  36
CS:25 833E000009 CMP  WORD PTR[0000],9
CS:2A 7D0A       JGE  36
CS:2C 8B0E0200   MOV  CX,[0002]
CS:30 010E0000   ADD  [0000],CX
CS:34 EBE3       JMP  19
```

```
Stony Brook QuickMod (24 bytes)

CS:69 BB2D00     MOV  BX,2D
CS:6C B90200     MOV  CX,2
CS:6F E90200     JMP  74
CS:72 01D9       ADD  CX,BX
CS:74 83F901     CMP  CX,1
CS:77 7F03       JG   7C
CS:79 E90500     JMP  81
CS:7C 83F909     CMP  CX,9
CS:7F 7CF1       JL   72
```

A translator inevitably makes use of a complex data structure, known as the **symbol table**, in which it keeps track of the names used by the program, and associated properties for these, such as their type, and their storage requirements (in the case of variables), or their values (in the case of constants).

As is well known, users of high-level languages are apt to make many errors in the development of even quite simple programs. Thus the various phases of a compiler, especially the earlier ones, also communicate with an **error handler** and **error reporter** which are invoked when errors are detected. It is desirable that compilation of erroneous programs be continued, if possible, so that the user can clean several errors out of the source before recompiling. This raises very interesting issues regarding the design of **error recovery** and **error correction** techniques. (We speak of error recovery when the translation process attempts to carry on after detecting an error, and of error correction or error repair when it attempts to correct the error from context - usually a contentious subject, as the correction may be nothing like what the programmer originally had in mind.)

Error detection at compile-time in the source code must not be confused with error detection at run-time when executing the object code. Many code generators are responsible for adding error-checking code to the object program (to check that subscripts for arrays stay in bounds, for example). This may be quite rudimentary, or it may involve adding considerable code and data structures for use with sophisticated debugging systems. Such ancillary code can drastically reduce the efficiency of a program, and some compilers allow it to be suppressed.

Sometimes mistakes in a program that are detected at compile-time are known as *errors*, and errors that show up at run-time are known as *exceptions*, but there is no universally agreed terminology for this.

Figure 2.4 seems to imply that compilers work serially, and that each phase communicates with the next by means of a suitable intermediate language, but in practice the distinction between the various phases often becomes a little blurred. Moreover, many compilers are actually constructed around a central parser as the dominant component, with a structure rather more like the one in Figure 2.7.

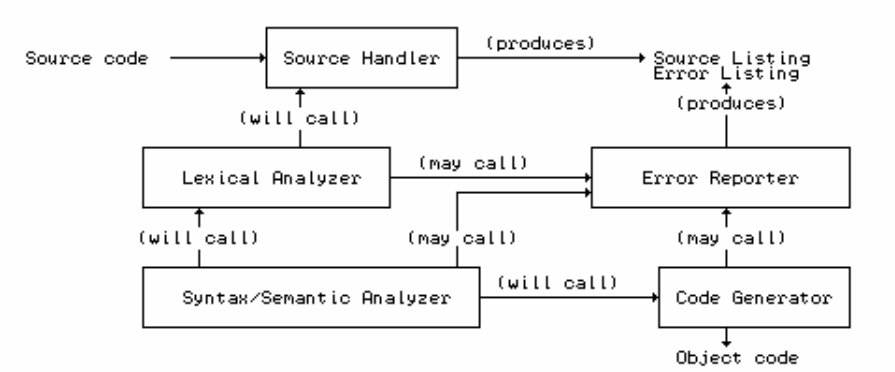


Figure 2.7 Structure of a parser-directed compiler

Exercises

2.5 What sort of problems can you foresee a Fortran compiler having in analysing statements beginning

```
      IF ( I(J) - I(K) ) ........
      CALL IF (4 ,    ...........
      IF (3 .EQ. MAX) GOTO ......
100   FORMAT(X3H)=(I5)
```

2.6 What sort of code would *you* have produced had you been coding a statement like "`WHILE (1 <`

`P) AND (P < 9) DO P := P + Q END`" into your favourite ASSEMBLER language?

2.7 Draw the concrete syntax tree for the C++ version of the *while* statement used for illustration in this section.

2.8 Are there any reasons why short-circuit evaluation should be preferred over the Boolean operator approach? Can you think of any algorithms that would depend critically on which approach was adopted?

2.9 Write down a few other high-level constructs and try to imagine what sort of ASSEMBLER-like machine code a compiler would produce for them.

2.10 What do you suppose makes it relatively easy to compile Pascal? Can you think of any aspects of Pascal which could prove really difficult?

2.11 We have used two undefined terms which at first seem interchangeable, namely "separate" and "independent" compilation. See if you can discover what the differences are.

2.12 Many development systems - in particular debuggers - allow a user to examine the object code produced by a compiler. If you have access to one of these, try writing a few very simple (single statement) programs, and look at the sort of object code that is generated for them.

2.4 Multi-stage translators

Besides being conceptually divided into phases, translators are often divided into **passes**, in each of which several phases may be combined or interleaved. Traditionally, a pass reads the source program, or output from a previous pass, makes some transformations, and then writes output to an intermediate file, whence it may be rescanned on a subsequent pass.

These passes may be handled by different integrated parts of a single compiler, or they may be handled by running two or more separate programs. They may communicate by using their own specialized forms of intermediate language, they may communicate by making use of internal data structures (rather than files), or they may make several passes over the same original source code.

The number of passes used depends on a variety of factors. Certain languages require at least two passes to be made if code is to be generated easily - for example, those where declaration of identifiers may occur after the first reference to the identifier, or where properties associated with an identifier cannot be readily deduced from the context in which it first appears. A multi-pass compiler can often save space. Although modern computers are usually blessed with far more memory than their predecessors of only a few years back, multiple passes may be an important consideration if one wishes to translate complicated languages within the confines of small systems. Multi-pass compilers may also allow for better provision of code optimization, error reporting and error handling. Lastly, they lend themselves to team development, with different members of the team assuming responsibility for different passes. However, multi-pass compilers are usually slower than single-pass ones, and their probable need to keep track of several files makes them slightly awkward to write and to use. Compromises at the design stage often result in languages that are well suited to single-pass compilation.

In practice, considerable use is made of two-stage translators in which the first stage is a high-level

translator that converts the source program into ASSEMBLER, or even into some other relatively high-level language for which an efficient translator already exists. The compilation process would then be depicted as in Figure 2.8 - our example shows a Modula-3 program being prepared for execution on a machine that has a Modula-3 to C converter:

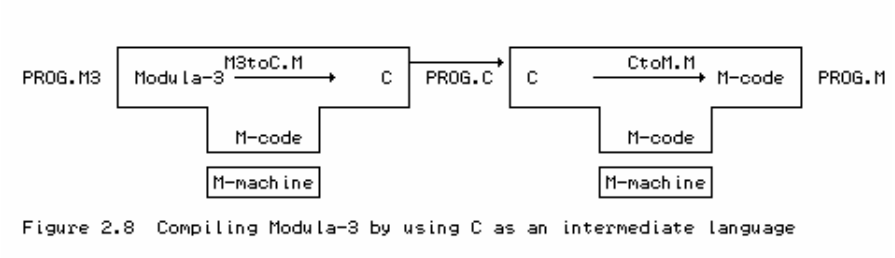


Figure 2.8 Compiling Modula-3 by using C as an intermediate language

It is increasingly common to find compilers for high-level languages that have been implemented using C, and which themselves produce C code as output. The success of these is based on the premises that "all modern computers come equipped with a C compiler" and "source code written in C is truly portable". Neither premise is, unfortunately, completely true. However, compilers written in this way are as close to achieving the dream of themselves being portable as any that exist at the present time. The way in which such compilers may be used is discussed further in Chapter 3.

Exercises

2.13 Try to find out which of the compilers you have used are single-pass, and which are multi-pass, and for the latter, find out how many passes are involved. Which produce relocatable code needing further processing by linkers or linkage editors?

2.14 Do any of the compilers in use on your system produce ASSEMBLER, C or other such code during the compilation process? Can you foresee any particular problems that users might experience in using such compilers?

2.15 One of several compilers that translates from Modula-2 to C is called `mtc`, and is freely available from several ftp sites. If you are a Modula-2 programmer, obtain a copy, and experiment with it.

2.16 An excellent compiler that translates Pascal to C is called `p2c`, and is widely available for Unix systems from several ftp sites. If you are a Pascal programmer, obtain a copy, and experiment with it.

2.17 Can you foresee any practical difficulties in using C as an intermediate language?

2.5 Interpreters, interpretive compilers, and emulators

Compilers of the sort that we have been discussing have a few properties that may not immediately be apparent. Firstly, they usually aim to produce object code that can run at the full speed of the target machine. Secondly, they are usually arranged to compile an entire section of code before any of it can be executed.

In some interactive environments the need arises for systems that can execute part of an application without preparing all of it, or ones that allow the user to vary his or her course of action on the fly. Typical scenarios involve the use of spreadsheets, on-line databases, or batch files or shell scripts for operating systems. With such systems it may be feasible (or even desirable) to exchange some of the advantages of speed of execution for the advantage of procuring results on demand.

Systems like these are often constructed so as to make use of an **interpreter**. An interpreter is a translator that effectively accepts a source program and executes it directly, without, seemingly, producing any object code first. It does this by fetching the source program instructions one by one, analysing them one by one, and then "executing" them one by one. Clearly, a scheme like this, if it is to be successful, places some quite severe constraints on the nature of the source program. Complex program structures such as nested procedures or compound statements do not lend themselves easily to such treatment. On the other hand, one-line queries made of a data base, or simple manipulations of a row or column of a spreadsheet, can be handled very effectively.

This idea is taken quite a lot further in the development of some translators for high-level languages, known as **interpretive compilers**. Such translators produce (as output) intermediate code which is intrinsically simple enough to satisfy the constraints imposed by a practical interpreter, even though it may still be quite a long way from the machine code of the system on which it is desired to execute the original program. Rather than continue translation to the level of machine code, an alternative approach that may perform acceptably well is to use the intermediate code as part of the input to a specially written interpreter. This in turn "executes" the original algorithm, by simulating a virtual machine for which the intermediate code effectively *is* the machine code. The distinction between the machine code and pseudo-code approaches to execution is summarized in Figure 2.9.

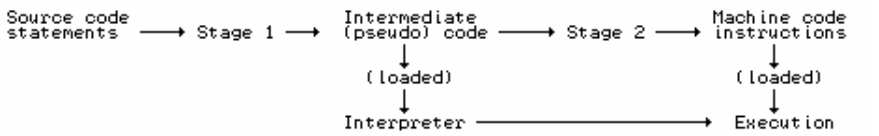


Figure 2.9 Differences between native-code and pseudo-code compilers

We may depict the process used in an interpretive compiler running under MS-DOS for a toy language like Clang, the one illustrated in later chapters, in T-diagram form (see Figure 2.10).

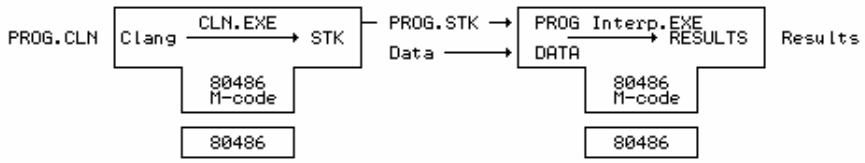


Figure 2.10 An interpretive compiler/interpreter for Clang

It is not necessary to confine interpreters merely to work with intermediate output from a translator. More generally, of course, even a real machine can be viewed as a highly specialized interpreter - one that executes the machine level instructions by fetching, analysing, and then interpreting them one by one. In a real machine this all happens "in hardware", and hence very quickly. By carrying on this train of thought, the reader should be able to see that a program could be written to allow one real machine to emulate any other real machine, albeit perhaps slowly, simply by writing an interpreter - or, as it is more usually called, an **emulator** - for the second machine.

For example, we might develop an emulator that runs on a Sun SPARC machine and makes it appear to be an IBM PC (or the other way around). Once we have done this, we are (in principle) in a position to execute any software developed for an IBM PC on the Sun SPARC machine - effectively the PC software becomes portable!

The T-diagram notation is easily extended to handle the concept of such virtual machines. For example, running Turbo Pascal on our Sun SPARC machine could be depicted by Figure 2.11.

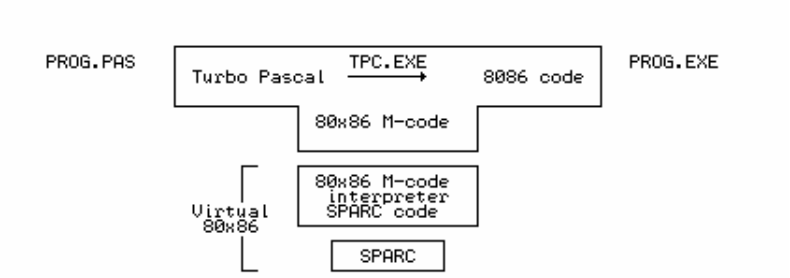


Figure 2.11 Executing the Turbo Pascal compiler on a Sun SPARC

The interpreter/emulator approach is widely used in the design and development both of new machines themselves, and the software that is to run on those machines.

An interpretive approach may have several points in its favour:

- It is far easier to generate hypothetical machine code (which can be tailored towards the quirks of the original source language) than real machine code (which has to deal with the uncompromising quirks of real machines).

- A compiler written to produce (as output) well-defined pseudo-machine code capable of easy interpretation on a range of machines can be made highly portable, especially if it is written in a host language that is widely available (such as ANSI C), or even if it is made available already implemented in its own pseudo- code.

- It can more easily be made "user friendly" than can the native code approach. Since the interpreter works closer to the source code than does a fully translated program, error messages and other debugging aids may readily be related to this source.

- A whole range of languages may quickly be implemented in a useful form on a wide range of different machines relatively easily. This is done by producing intermediate code to a well-defined standard, for which a relatively efficient interpreter should be easy to implement on any particular real machine.

- It proves to be useful in connection with cross-translators such as were mentioned earlier. The code produced by such translators can sometimes be tested more effectively by simulated execution on the donor machine, rather than after transfer to the target machine - the delays inherent in the transfer from one machine to the other may be balanced by the degradation of execution time in an interpretive simulation.

- Lastly, intermediate languages are often very compact, allowing large programs to be handled, even on relatively small machines. The success of the once very widely used UCSD Pascal and UCSD p-System stands as an example of what can be done in this respect.

For all these advantages, interpretive systems carry fairly obvious overheads in execution speed, because execution of intermediate code effectively carries with it the cost of virtual translation into machine code each time a hypothetical machine instruction is obeyed.

One of the best known of the early **portable interpretive compilers** was the one developed at Zürich and known as the "Pascal-P" compiler (Nori *et al.*, 1981). This was supplied in a kit of three components:

- The first component was the source form of a Pascal compiler, written in a very complete subset of the language, known as Pascal-P. The aim of this compiler was to translate Pascal-P source programs into a well-defined and well-documented intermediate language, known as P-code, which was the "machine code" for a hypothetical stack-based computer, known as the P-machine.

- The second component was a compiled version of the first - the P-codes that would be produced by the Pascal-P compiler, were it to compile itself.

- Lastly, the kit contained an interpreter for the P-code language, supplied as a Pascal algorithm.

The interpreter served primarily as a model for writing a similar program for the target machine, to allow it to emulate the hypothetical P-machine. As we shall see in a later chapter, emulators are relatively easy to develop - even, if necessary, in ASSEMBLER - so that this stage was usually fairly painlessly achieved. Once one had loaded the interpreter - that is to say, the version of it tailored to a local real machine - into a real machine, one was in a position to "execute" P-code, and in particular the P-code of the P-compiler. The compilation and execution of a user program could then be achieved in a manner depicted in Figure 2.12.

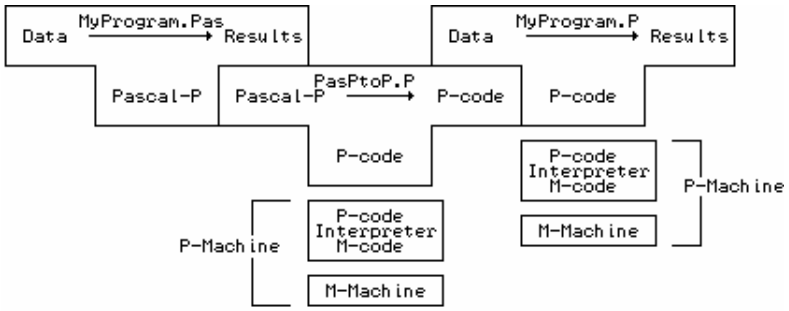


Figure 2.12 Compiling and executing a program with the P-compiler

---

Exercises

2.18 Try to find out which of the translators you have used are interpreters, rather than full compilers.

2.19 If you have access to both a native-code compiler and an interpreter for a programming language known to you, attempt to measure the loss in efficiency when the interpreter is used to run a large program (perhaps one that does substantial number-crunching).

3 COMPILER CONSTRUCTION AND BOOTSTRAPPING

By now the reader may have realized that developing translators is a decidedly non-trivial exercise. If one is faced with the task of writing a full-blown translator for a fairly complex source language, or an emulator for a new virtual machine, or an interpreter for a low-level intermediate language, one would probably prefer not to implement it all in machine code.

Fortunately one rarely has to contemplate such a radical step. Translator systems are now widely available and well understood. A fairly obvious strategy when a translator is required for an old language on a new machine, or a new language on an old machine (or even a new language on a new machine), is to make use of existing compilers on either machine, and to do the development in a high level language. This chapter provides a few examples that should make this clearer.

3.1 Using a high-level host language

If, as is increasingly common, one's dream machine *M* is supplied with the machine coded version of a compiler for a well-established language like C, then the production of a compiler for one's dream language *X* is achievable by writing the new compiler, say *XtoM*, in C and compiling the source (*XtoM.C*) with the C compiler (*CtoM.M*) running directly on *M* (see Figure 3.1). This produces the object version (*XtoM.M*) which can then be executed on *M*.

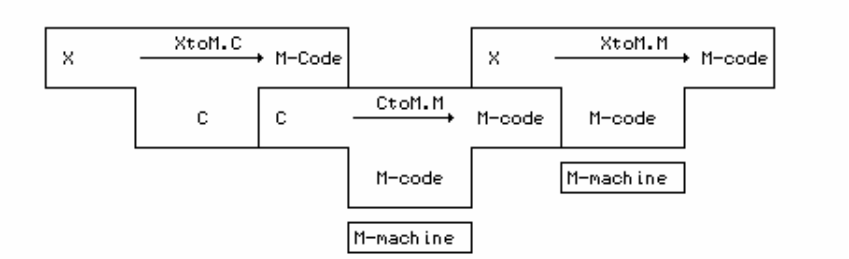


Figure 3.1 Use of C as an implementation language

Even though development in C is much easier than development in machine code, the process is still complex. As was mentioned earlier, it may be possible to develop a large part of the compiler source using compiler generator tools - assuming, of course, that these are already available either in executable form, or as C source that can itself be compiled easily. The hardest part of the development is probably that associated with the back end, since this is intensely machine dependent. If one has access to the source code of a compiler like *CtoM* one may be able to use this to good avail. Although commercial compilers are rarely released in source form, source code is available for many compilers produced at academic institutions or as components of the GNU project carried out under the auspices of the Free Software Foundation.

3.2 Porting a high level translator

The process of modifying an existing compiler to work on a new machine is often known as

porting the compiler. In some cases this process may be almost trivially easy. Consider, for example, the fairly common scenario where a compiler *XtoC* for a popular language *X* has been implemented in C on machine *A* by writing a high-level translator to convert programs written in *X* to C, and where it is desired to use language *X* on a machine *M* that, like *A*, has already been blessed with a C compiler of its own. To construct a two-stage compiler for use on either machine, all one needs to do, in principle, is to install the source code for *XtoC* on machine *M* and recompile it.

Such an operation is conveniently represented in terms of T-diagrams chained together. Figure 3.2(a) shows the compilation of the *X* to C compiler, and Figure 3.2(b) shows the two-stage compilation process needed to compile programs written in *X* to *M*-code.

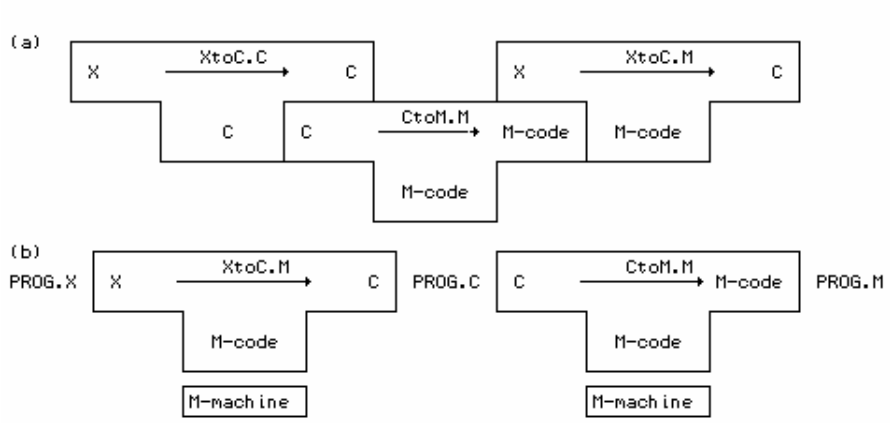


Figure 3.2 Porting and using a high-level translator

The portability of a compiler like *XtoC.C* is almost guaranteed, provided that it is itself written in "portable" C. Unfortunately, or as Mr. Murphy would put it, "interchangeable parts don’t" (more explicitly, "portable C isn’t"). Some time may have to be spent in modifying the source code of *XtoC.C* before it is acceptable as input to *CtoM.M*, although it is to be hoped that the developers of *XtoC.C* will have used only standard C in their work, and used pre-processor directives that allow for easy adaptation to other systems.

If there is an initial strong motivation for making a compiler portable to other systems it is, indeed, often written so as to produce high-level code as output. More often, of course, the original implementation of a language is written as a self-resident translator with the aim of directly producing machine code for the current host system.

3.3 Bootstrapping

All this may seem to be skirting around a really nasty issue - how might the first high-level language have been implemented? In ASSEMBLER? But then how was the assembler for ASSEMBLER produced?

A full assembler is itself a major piece of software, albeit rather simple when compared with a compiler for a really high level language, as we shall see. It is, however, quite common to define one language as a subset of another, so that subset 1 is contained in subset 2 which in turn is contained in subset 3 and so on, that is:



One might first write an assembler for subset 1 of ASSEMBLER in machine code, perhaps on a load-and-go basis (more likely one writes in ASSEMBLER, and then hand translates it into machine code). This subset assembler program might, perhaps, do very little other than convert mnemonic opcodes into binary form. One might then write an assembler for subset 2 of ASSEMBLER in subset 1 of ASSEMBLER, and so on.

This process, by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is known as **bootstrapping**, by analogy with the idea that it might be possible to lift oneself off the ground by tugging at one's boot-straps.

3.4 Self-compiling compilers

Once one has a working system, one can start using it to improve itself. Many compilers for popular languages were first written in another implementation language, as implied in section 3.1, and then rewritten in their own source language. The rewrite gives source for a compiler that can then be compiled with the compiler written in the original implementation language. This is illustrated in Figure 3.3.

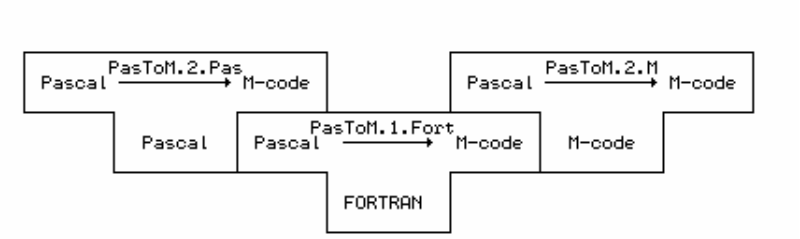


Figure 3.3 First step in developing a self-compiling compiler

Clearly, writing a compiler by hand not once, but twice, is a non-trivial operation, unless the original implementation language is close to the source language. This is not uncommon: Oberon compilers could be implemented in Modula-2; Modula-2 compilers, in turn, were first implemented in Pascal (all three are fairly similar), and C++ compilers were first implemented in C.

Developing a self-compiling compiler has four distinct points to recommend it. Firstly, it constitutes a non-trivial test of the viability of the language being compiled. Secondly, once it has been done, further development can be done without recourse to other translator systems. Thirdly, any improvements that can be made to its back end manifest themselves both as improvements to the object code it produces for general programs and as improvements to the compiler itself. Lastly, it provides a fairly exhaustive self-consistency check, for if the compiler is used to compile its own source code, it should, of course, be able to reproduce its own object code (see Figure 3.4).

Furthermore, given a working compiler for a high-level language it is then very easy to produce compilers for specialized dialects of that language.

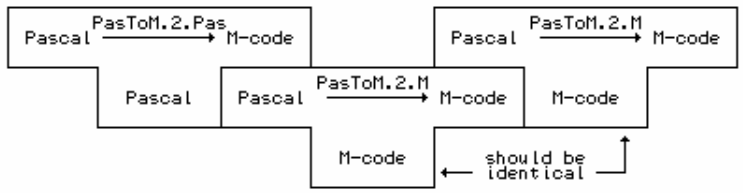


Figure 3.4 A self-compiling compiler must be self-consistent

3.5 The half bootstrap

Compilers written to produce object code for a particular machine are not intrinsically portable. However, they are often used to assist in a porting operation. For example, by the time that the first Pascal compiler was required for ICL machines, the Pascal compiler available in Zürich (where Pascal had first been implemented on CDC mainframes) existed in two forms (Figure 3.5).



Figure 3.5 Two versions of the original Zürich Pascal compiler

The first stage of the transportation process involved changing *PasToCDC.Pas* to generate ICL machine code - thus producing a cross compiler. Since *PasToCDC.Pas* had been written in a high-level language, this was not too difficult to do, and resulted in the compiler *PasToICL.Pas*.

Of course this compiler could not yet run on any machine at all. It was first compiled using *PasToCDC.CDC*, on the CDC machine (see Figure 3.6(a)). This gave a cross-compiler that could run on CDC machines, but still not, of course, on ICL machines. One further compilation of *PasToICL.Pas*, using the cross-compiler *PasToICL.CDC* on the CDC machine, produced the final result, *PasToICL.ICL* (Figure 3.6(b)).

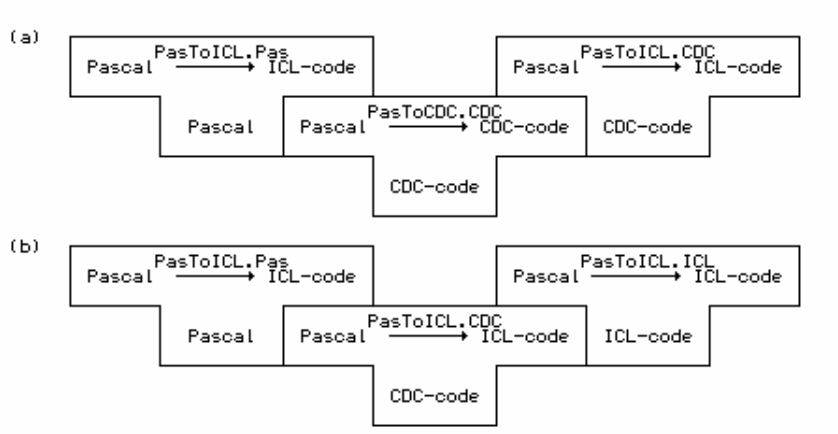


Figure 3.6 Production of the first ICL Pascal compiler by half-bootstrap

The final product (*PasToICL.ICL*) was then transported on magnetic tape to the ICL machine, and loaded quite easily. Having obtained a working system, the ICL team could (and did) continue development of the system in Pascal itself.

This porting operation was an example of what is known as a *half bootstrap* system. The work of transportation is essentially done entirely on the donor machine, without the need for any translator in the target machine, but a crucial part of the original compiler (the back end, or code generator) has to be rewritten in the process. Clearly the method is hazardous - any flaws or oversights in writing *PasToICL.Pas* could have spelled disaster. Such problems can be reduced by minimizing changes made to the original compiler. Another technique is to write an emulator for the target machine that runs on the donor machine, so that the final compiler can be tested on the donor machine before being transferred to the target machine.

3.6 Bootstrapping from a portable interpretive compiler

Because of the inherent difficulty of the half bootstrap for porting compilers, a variation on the full bootstrap method described above for assemblers has often been successfully used in the case of Pascal and other similar high-level languages. Here most of the development takes place on the target machine, after a lot of preliminary work has been done on the donor machine to produce an interpretive compiler that is almost portable. It will be helpful to illustrate with the well-known example of the Pascal-P implementation kit mentioned in section 2.5.

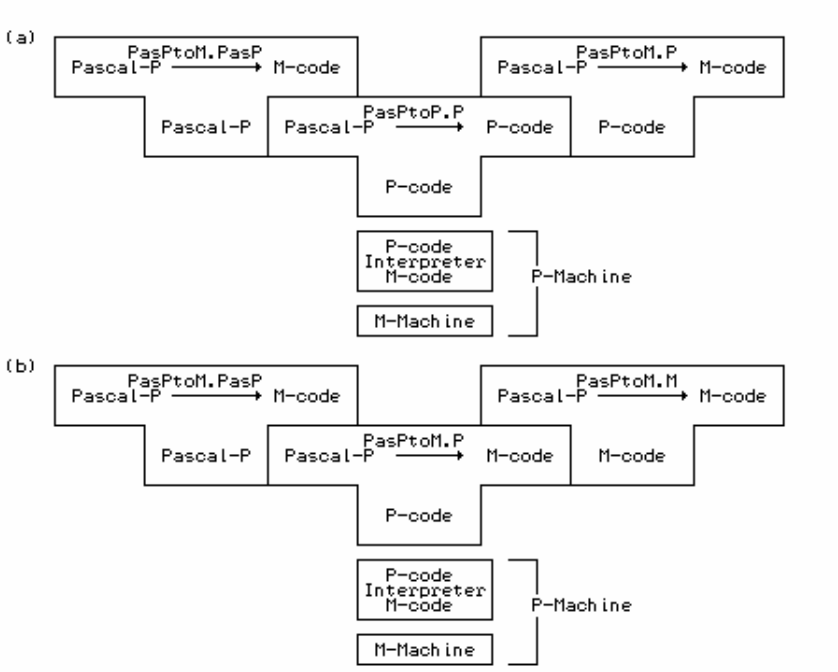


Figure 3.7 Developing a native code compiler from the P-compiler

Users of this kit typically commenced operations by implementing an interpreter for the P-machine. The bootstrap process was then initiated by developing a compiler (*PasPtoM.PasP*) to translate Pascal-P source programs to the local machine code. This compiler could be written in Pascal-P source, development being guided by the source of the Pascal-P to P-code compiler supplied as part of the kit. This new compiler was then compiled with the interpretive compiler (*PasPtoP.P*) from the kit (Figure 3.7(a)) and the source of the Pascal to M-code compiler was then compiled by this

new compiler, interpreted once again by the P-machine, to give the final product, *PasPtoM.M* (Figure 3.7(b)).

The Zürich P-code interpretive compiler could be, and indeed was, used as a highly portable development system. It was employed to remarkable effect in developing the UCSD Pascal system, which was the first serious attempt to implement Pascal on microcomputers. The UCSD Pascal team went on to provide the framework for an entire operating system, editors and other utilities - all written in Pascal, and all compiled into a well-defined P-code object code. Simply by providing an alternative interpreter one could move the whole system to a new microcomputer system virtually unchanged.

3.7 A P-code assembler

There is, of course, yet another way in which a portable interpretive compiler kit might be used. One might commence by writing a P-code to M-code assembler, probably a relatively simple task. Once this has been produced one would have the assembler depicted in Figure 3.8.

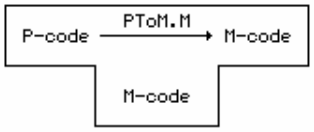


Figure 3.8 A P-code to M-code assembler

The P-codes for the P-code compiler would then be assembled by this system to give another cross compiler (Figure 3.9(a)), and the same P-code/M-code assembler could then be used as a back-end to the cross compiler (Figure 3.9(b)).

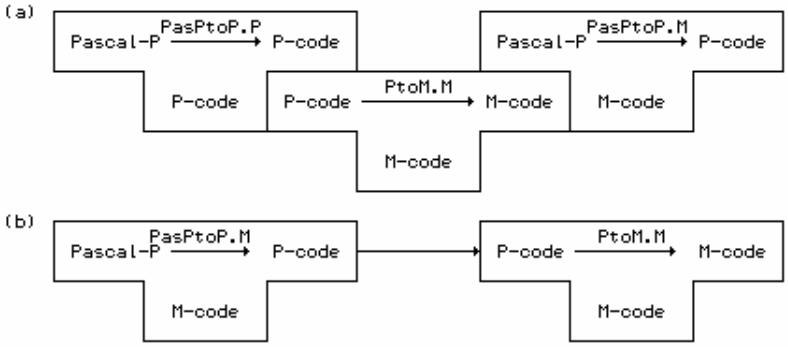


Figure 3.9 Two-pass compilation and assembly using a P-code compiler

---

Exercises

3.1 Draw the T-diagram representations for the development of a P-code to M-code assembler, assuming that you have a C++ compiler available on the target system.

3.2 Later in this text we shall develop an interpretive compiler for a small language called Clang,

using C++ as the host language. Draw T-diagram representations of the various components of the system as you foresee them.

Further reading

A very clear exposition of bootstrapping is to be found in the book by Watt (1993). The ICL bootstrap is further described by Welsh and Quinn (1972). Other early insights into bootstrapping are to be found in papers by Lecarme and Peyrolle-Thomas (1973), by Nori *et al.* (1981), and Cornelius, Lowman and Robson (1984).

4 MACHINE EMULATION

In Chapter 2 we discussed the use of emulation or interpretation as a tool for programming language translation. In this chapter we aim to discuss hypothetical machine languages and the emulation of hypothetical machines for these languages in more detail. Modern computers are among the most complex machines ever designed by the human mind. However, this is a text on programming language translation and not on electronic engineering, and our restricted discussion will focus only on rather primitive object languages suited to the simple translators to be discussed in later chapters.

4.1 Simple machine architecture

Many CPU (central processor unit) chips used in modern computers have one or more internal **registers** or **accumulators**, which may be regarded as highly local memory where simple arithmetic and logical operations may be performed, and between which local data transfers may take place. These registers may be restricted to the capacity of a single byte (8 bits), or, as is typical of most modern processors, they may come in a variety of small multiples of bytes or machine words.

One fundamental internal register is the **instruction register** (IR), through which moves the bitstrings (bytes) representing the fundamental machine-level instructions that the processor can obey. These instructions tend to be extremely simple - operations such as "clear a register" or "move a byte from one register to another" being the typical order of complexity. Some of these instructions may be completely defined by a single byte value. Others may need two or more bytes for a complete definition. Of these multi-byte instructions, the first usually denotes an operation, and the rest relate either to a value to be operated upon, or to the address of a location in memory at which can be found the value to be operated upon.

The simplest processors have only a few **data registers**, and are very limited in what they can actually do with their contents, and so processors invariably make provision for interfacing to the memory of the computer, and allow transfers to take place along so-called **bus** lines between the internal registers and the far greater number of external memory locations. When information is to be transferred to or from memory, the CPU places the appropriate address information on the address bus, and then transmits or receives the data itself on the data bus. This is illustrated in Figure 4.1.

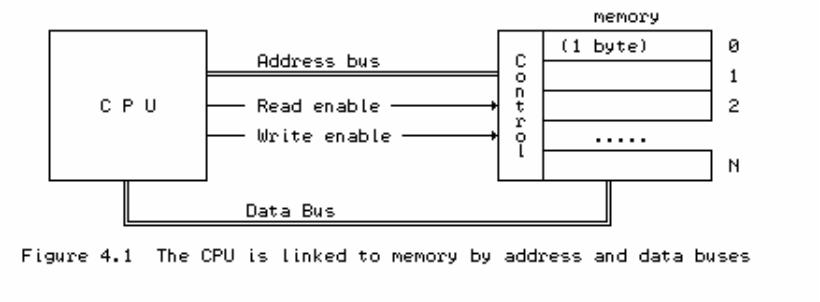


Figure 4.1 The CPU is linked to memory by address and data buses

The memory may simplistically be viewed as a one-dimensional array of byte values, analogous to what might be described in high-level language terms by declarations like the following

```
TYPE
  ADDRESS = CARDINAL [0 .. MemSize - 1];
  BYTES   = CARDINAL [0 .. 255];
VAR
  Mem : ARRAY ADDRESS OF BYTES;
```

in Modula-2, or, in C++ (which does not provide for the subrange types so useful in this regard)

```
typedef unsigned char BYTES;
BYTES Mem[MemSize];
```

Since the memory is used to store not only "data" but also "instructions", another important internal register in a processor, the so-called **program counter** or **instruction pointer** (denoted by PC or IP), is used to keep track of the address in memory of the next instruction to be fed to the processor's instruction register (IR).

Perhaps it will be helpful to think of the processor itself in high-level terms:

```
TYPE
  PROCESSOR =                   struct processor {
    RECORD                        BYTES IR;
      IR,                         BYTES R1, R2, R3;
      R1, R2, R3 : BYTES;         unsigned PC;
      PC : ADDRESS;             };
    END;
VAR                             processor cpu;
  CPU : PROCESSOR;
```

The operation of the machine is repeatedly to *fetch* a byte at a time from memory (along the data bus), place it in the IR, and then *execute* the operation which this byte represents. Multi-byte instructions may require the fetching of further bytes before the instruction itself can be decoded fully by the CPU, of course. After the instruction denoted by the contents of IR has been executed, the value of PC will have been changed to point to the next instruction to be fetched. This **fetch-execute cycle** may be described by the following algorithm:

```
BEGIN
  CPU.PC := initialValue;  (* address of first code instruction *)
  LOOP
    CPU.IR := Mem[CPU.PC]; (* fetch *)
    Increment(CPU.PC);     (* bump PC in anticipation *)
    Execute(CPU.IR);       (* affecting other registers, memory, PC *)
                           (* handle machine interrupts if necessary *)
  END
END.
```

Normally the value of PC alters by small steps (since instructions are usually stored in memory in sequence); execution of branch instructions may, however, have a rather more dramatic effect. So might the occurrence of hardware interrupts, although we shall not discuss interrupt handling further.

A program for such a machine consists, in the last resort, of a long string of byte values. Were these to be written on paper (as binary, decimal, or hexadecimal values), they would appear pretty meaningless to the human reader. We might, for example, find a section of program reading

```
        25  45  21  34  34  30  45
```

Although it may not be obvious, this might be equivalent to a high-level statement like

```
        Price := 2 * Price + MarkUp;
```

Machine-level programming is usually performed by associating *mnemonics* with the recognizable

operations, like `HLT` for "halt" or `ADD` for "add to register". The above code is far more comprehensible when written (with commentary) as

```
        LDA  45   ; load accumulator with value stored in memory location 45
        SHL       ; shift accumulator one bit left (multiply by 2)
        ADI  34   ; add 34 to the accumulator
        STA  45   ; store the value in the accumulator at memory location 45
```

Programs written in an assembly language - which have first to be assembled before they can be executed - usually make use of other named entities, for example

```
MarkUp EQU  34       ; CONST MarkUp = 34;
       LDA  Price    ; CPU.A := Price;
       SHL           ; CPU.A := 2 * CPU.A;
       ADI  MarkUp   ; CPU.A := CPU.A + 34;
       STA  Price    ; Price := CPU.A;
```

When we use code fragments such as these for illustration we shall make frequent use of commentary showing an equivalent fragment written in a high-level language. Commentary follows the semicolon on each line, a common convention in assembler languages.

4.2 Addressing modes

As the examples given earlier suggest, programs prepared at or near the machine level frequently consist of a sequence of simple instructions, each involving a machine-level operation and one or more parameters.

An example of a simple operation expressed in a high-level language might be

```
AmountDue := Price + Tax;
```

Some machines and assembler languages provide for such operations in terms of so-called **three-address code**, in which an *operation* - denoted by a mnemonic usually called the **opcode** - is followed by two *operands* and a *destination*. In general this takes the form

operation destination, operand$_1$*, operand*$_2$

for example

```
ADD      AmountDue, Price, Tax
```

We may also express this in a general sense as a function call

destination := operation(operand$_1$*, operand*$_2$ *)*

which helps to stress the important idea that the *operands* really denote "values", while the *destination* denotes a processor register, or an address in memory where the result is to be stored.

In many cases this generality is restricted (that is, the machine suffers from non-orthogonality in design). Typically the value of one *operand* is required to be the value originally stored at the *destination*. This corresponds to high-level statements like

```
Price := Price * InflationFactor;
```

and is mirrored at the low-level by so-called **two-address code** of the general form

operation destination, operand

for example

```
MUL      Price, InflationFactor
```

In passing, we should point out an obvious connection between some of the assignment operations in C++ and two-address code. In C++ the above assignment would probably have been written

```
Price *= InflationFactor;
```

which, while less transparent to a Modula-2 programmer, is surely a hint to a C++ compiler to generate code of this form. (Perhaps this example may help you understand why C++ is regarded by some as the world's finest assembly language!)

In many real machines even general two-address code is not found at the machine level. One of *destination* and *operand* might be restricted to denoting a machine register (the other one might denote a machine register, or a constant, or a machine address). This is often called **one and a half address code**, and is exemplified by

```
MOV     R1, Value     ; CPU.R1 := Value
ADD     Answer, R1    ; Answer := Answer + CPU.R1
MOV     Result, R2    ; Result := CPU.R2
```

Finally, in so-called *accumulator machines* we may be restricted to **one-address code**, where the destination is always a machine register (except for those operations that copy (store) the contents of a machine register into memory). In some assembler languages such instructions may still appear to be of the two-address form, as above. Alternatively they might be written in terms of opcodes that have the register implicit in the mnemonic, for example

```
LDA     Value        ; CPU.A := Value
ADA     Answer       ; CPU.A := CPU.A + Answer
STB     Result       ; Result := CPU.B
```

Although many of these examples might give the impression that the corresponding machine level operations require multiple bytes for their representation, this is not necessarily true. For example, operations that only involve machine registers, exemplified by

```
MOV     R1, R2       ;  CPU.R1 := CPU.R2
LDA     B            ;  CPU.A := CPU.B
TAX                  ;  CPU.X := CPU.A
```

might require only a single byte - as would be most obvious in an assembler language that used the third representation. The assembly of such programs is be eased considerably by a simple and self-consistent notation for the source code, a subject that we shall consider further in a later chapter.

In those instructions that *do* involve the manipulation of values other than those in the machine registers alone, multi-byte instructions are usually required. The first byte typically specifies the operation itself (and possibly the register or registers that are involved), while the remaining bytes specify the other values (or the memory addresses of the other values) involved. In such instructions there are several ways in which the ancillary bytes might be used. This variety gives rise to what are known as different **addressing modes** for the processor, and whose purpose it is to provide an **effective address** to be used in an instruction. Exactly which modes are available varies tremendously from processor to processor, and we can mention only a few representative examples here. The various possibilities may be distinguished in some assembler languages by the use of different mnemonics for what at first sight appear to be closely related operations. In other assembler languages the distinction may be drawn by different syntactic forms used to specify the registers, addresses or values. One may even find different assembler languages for a common

processor.

In **inherent addressing** the operand is implicit in the opcode itself, and often the instruction is contained in a single byte. For example, to clear a machine register named `A` we might have

```
CLA          or       CLR  A         ;  CPU.A := 0
```

Again we stress that, though the second form seems to have two components, it does not always imply the use of two bytes of code at the machine level.

In **immediate addressing** the ancillary bytes for an instruction typically give the *actual value* that is to be combined with a value in a register. Examples might be

```
ADI  34      or       ADD  A, #34    ;  CPU.A := CPU.A + 34
```

In these two addressing modes the use of the word "address" is almost misleading, as the value of the ancillary bytes may often have nothing to do with a memory address at all. In the modes now to be discussed the connection with memory addresses is far more obvious.

In **direct** or **absolute addressing** the ancillary bytes typically specify the *memory address* of the value that is to be retrieved or combined with the value in a register, or specify where a register value is to be stored. Examples are

```
LDA  34      or       MOV  A, 34     ;  CPU.A := Mem[34]
STA  45               MOV  45, A     ;  Mem[45] := CPU.A
ADD  38               ADD  A, 38     ;  CPU.A := CPU.A + Mem[38]
```

Beginners frequently confuse immediate and direct addressing, a situation not improved by the fact that there is no consistency in notation between different assembler languages, and there may even be a variety of ways of expressing a particular addressing mode. For example, for the Intel 80x86 processors as used in the IBM-PC and compatibles, low-level code is written in a two-address form similar to that shown above - but the immediate mode is denoted without needing a special symbol like #, while the direct mode may have the address in brackets:

```
ADD  AX, 34    ;  CPU.AX := CPU.AX + 34  Immediate
MOV  AX, [34]  ;  CPU.AX := Mem[34]      Direct
```

In **register-indexed addressing** one of the operands in an instruction specifies both an address and also an *index register*, whose value at the time of execution may be thought of as specifying the subscript to an array stored from that address

```
LDX  34      or       MOV  A, 34[X]  ;  CPU.A := Mem[34 + CPU.X]
STX  45               MOV  45[X], A  ;  Mem[45+CPU.X] := CPU.A
ADX  38               ADD  A, 38[X]  ;  CPU.A := CPU.A + Mem[38+CPU.X]
```

In **register-indirect addressing** one of the operands in an instruction specifies a register whose value at the time of execution gives the effective address where the value of the operand is to be found. This relates to the concept of *pointers* as used in Modula-2, Pascal and C++.

```
MOV   R1, @R2     ;  CPU.R1 := Mem[CPU.R2]
MOV   AX, [BX]    ;  CPU.AX := Mem[CPU.BX]
```

Not all the registers in a machine can necessarily be used in these ways. Indeed, some machines have rather awkward restrictions in this regard.

Some processors allow for very powerful variations on indexed and indirect addressing modes. For example, in **memory-indexed** addressing, a single operand may specify two memory addresses - the first of which gives the address of the first element of an array, and the second of which gives

the address of a variable whose value will be used as a subscript to the array.

```
        MOV   R1, 400[100]    ;  CPU.R1 := Mem[400 + Mem[100]]
```

Similarly, in **memory-indirect addressing** one of the operands in an instruction specifies a memory address at which will be found a value that forms the effective address where another operand is to be found.

```
        MOV   R1, @100    ;  CPU.R1 := Mem[Mem[100]]
```

This mode is not as commonly found as the others; where it does occur it directly corresponds to the use of pointer variables in languages that support them. Code like

```
TYPE
  ARROW = POINTER TO CARDINAL;          typedef int *ARROW;
VAR
  Arrow  : ARROW;                       ARROW Arrow;
  Target : CARDINAL;                    int Target;
BEGIN
  Target := Arrow^;                     Target = *Arrow;
```

might translate to equivalent code in assembler like

```
        MOV   AX, @Arrow
        MOV   Target, AX
```

or even

```
        MOV   Target, @Arrow
```

where, once again, we can see an immediate correspondence between the syntax in C++ and the corresponding assembler.

Finally, in **relative addressing** an operand specifies an amount by which the current program count register PC must be incremented or decremented to find the actual address of interest. This is chiefly found in "branching" instructions, rather than in those that move data between various registers and/or locations in memory.

Further reading

Most books on assembler level programming have far deeper discussions of the subject of addressing modes than we have presented. Two very readable accounts are to be found in the books by Wakerly (1981) and MacCabe (1993). A deeper discussion of machine architectures is to be found in the book by Hennessy and Patterson (1990).

4.3 Case study 1 - A single-accumulator machine

Although sophisticated processors may have several registers, their basic principles - especially as they apply to emulation - may be illustrated by the following model of a single-accumulator processor and computer, very similar to one suggested by Wakerly (1981). Here we shall take things to extremes and presume the existence of a system with all registers only 1 byte (8 bits) wide.

4.3.1 Machine architecture

Diagrammatically we might represent this machine as in Figure 4.2.

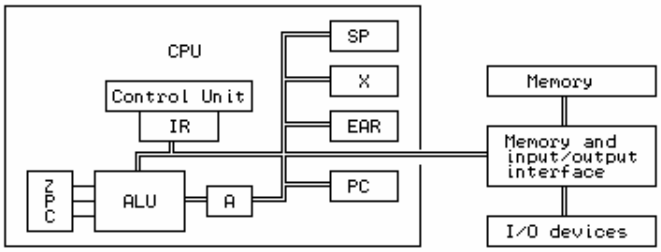


Figure 4.2 A simple single-accumulator CPU and computer

The symbols in this diagram refer to the following components of the machine

> `ALU` is the *arithmetic logic unit*, where arithmetic and logical operations are actually performed.
>
> `A` is the 8-bit *accumulator*, a register for doing arithmetic or logical operations.
>
> `SP` is an 8-bit *stack pointer*, a register that points to an area in memory that may be utilized as a stack.
>
> `X` is an 8-bit *index register*, which is used in indexing areas of memory which conceptually form data arrays.
>
> `Z, P, C` are single bit *condition flags* or *status registers*, which are set "true" when an operation causes a register to change to a zero value, or to a positive value, or to propagate a carry, respectively.
>
> `IR` is the 8-bit *instruction register*, in which is held the byte value of the instruction currently being executed.
>
> `PC` is the 8-bit *program counter*, which contains the address in memory of the instruction that is next to be executed.
>
> `EAR` is the *effective address register*, which contains the address of the byte of data which is being manipulated by the current instruction.

The programmer's model of this sort of machine is somewhat simpler - it consists of a number of "variables" (in the C++ or Modula-2 sense), each of which is one byte in capacity. Some of these correspond to processor registers, while the others form the random access read/write (RAM) memory, of which we have assumed there to be 256 bytes, addressed by the values 0 through 255. In this memory, as usual, will be stored both the data and the instructions for the program under execution. The processor, its registers, and the associated RAM memory can be thought of as though they were described by declarations like

```
TYPE
  BYTES = CARDINAL [0 .. 255];
  PROCESSOR = RECORD
    A, SP, X, IR, PC : BYTES;
    Z, P, C : BOOLEAN;
  END;
```

```
typedef unsigned char bytes;
struct processor {
  bytes a, sp, x, ir, pc;
  bool z, p, c;
};
```

```
TYPE STATUS = (running, finished,              typedef enum { running, finished,
              nodata, baddata,                   nodata, baddata, badop
              badop);                          } status;
VAR
  CPU : PROCESSOR;                             processor cpu;
  Mem : ARRAY BYTES OF BYTES;                  bytes mem[256];
  PS  : STATUS;                                status ps;
```

where the concept of the **processor status** PS has been introduced in terms of an enumeration that defines the states in which an emulator might find itself.

4.3.2 Instruction set

Some machine operations are described by a single byte. Others require two bytes, and have the format

```
Byte 1     Opcode
Byte 2     Address field
```

The set of machine code functions available is quite small. Those marked * affect the P and Z flags, and those marked + affect the C flag. An informal description of their semantics follows:

Mnemonic			Hex opcode	Decimal	Function
NOP			00h	0	No operation (this might be used to set a break point in an emulator)
CLA			01h	1	Clear accumulator A
CLC	+		02h	2	Clear carry bit C
CLX			03h	3	Clear index register X
CMC	+		04h	4	Complement carry bit C
INC		*	05h	5	Increment accumulator A by 1
DEC		*	06h	6	Decrement accumulator A by 1
INX		*	07h	7	Increment index register X by 1
DEX		*	08h	8	Decrement index register X by 1
TAX			09h	9	Transfer accumulator A to index register X
INI		*	0Ah	10	Load accumulator A with integer read from input in decimal
INH		*	0Bh	11	Load accumulator A with integer read from input in hexadecimal
INB		*	0Ch	12	Load accumulator A with integer read from input in binary
INA		*	0Dh	13	Load accumulator A with ASCII value read from input (a single character)
OTI			0Eh	14	Write value of accumulator A to output as a signed decimal number
OTC			0Fh	15	Write value of accumulator A to output as an unsigned decimal number
OTH			10h	16	Write value of accumulator A to output as an unsigned hexadecimal number
OTB			11h	17	Write value of accumulator A to output as an unsigned binary number
OTA			12h	18	Write value of accumulator A to output as a single character
PSH			13h	19	Decrement SP and push value of accumulator A onto stack
POP		*	14h	20	Pop stack into accumulator A and increment SP
SHL	+	*	15h	21	Shift accumulator A one bit left
SHR	+	*	16h	22	Shift accumulator A one bit right
RET			17h	23	Return from subroutine (return address popped from stack)
HLT			18h	24	Halt program execution

The above are all single-byte instructions. The following are all double-byte instructions.

LDA	B	*	19h	25	Load accumulator A directly with contents of location whose address is given as B
LDX	B	*	1Ah	26	Load accumulator A with contents of location whose address is given as B, indexed by the value of X (that is, an address computed as the value of B + X)
LDI	B	*	1Bh	27	Load accumulator A with the immediate value B
LSP	B		1Ch	28	Load stack pointer SP with contents of location whose address is given as B
LSI	B		1Dh	29	Load stack pointer SP immediately with the value B
STA	B		1Eh	30	Store accumulator A on the location whose address is given as B

STX	B		1Fh	31	Store accumulator A on the location whose address is given as B, indexed by the value of X
ADD	B	+ *	20h	32	Add to accumulator A the contents of the location whose address is given as B
ADX	B	+ *	21h	33	Add to accumulator A the contents of the location whose address is given as B,indexed by the value of X
ADI	B	+ *	22h	34	Add the immediate value B to accumulator A
ADC	B	+ *	23h	35	Add to accumulator A the value of the carry bit C plus the contents of the location whose address is given as B
ACX	B	+ *	24h	36	Add to accumulator A the value of the carry bit C plus the contents of the location whose address is given as B, indexed by the value of X
ACI	B	+ *	25h	37	Add the immediate value B plus the value of the carry bit C to accumulator A
SUB	B	+ *	26h	38	Subtract from accumulator A the contents of the location whose address is given as B
SBX	B	+ *	27h	39	Subtract from accumulator A the contents of the location whose address is given as B, indexed by the value of X
SBI	B	+ *	28h	40	Subtract the immediate value B from accumulator A
SBC	B	+ *	29h	41	Subtract from accumulator A the value of the carry bit C plus the contents of the location whose address is given as B
SCX	B	+ *	2Ah	42	Subtract from accumulator A the value of the carry bit C plus the contents of the location whose address is given as B, indexed by the value of X
SCI	B	+ *	2Bh	43	Subtract the immediate value B plus the value of the carry bit C from accumulator A
CMP	B	+ *	2Ch	44	Compare accumulator A with the contents of the location whose address is given as B
CPX	B	+ *	2Dh	45	Compare accumulator A with the contents of the location whose address is given as B, indexed by the value of X
CPI	B	+ *	2Eh	46	Compare accumulator A directly with the value B

These comparisons are done by virtual subtraction of the operand from A, and setting the flags P and Z as appropriate

ANA	B	+ *	2Fh	47	Bitwise AND accumulator A with the contents of the location whose address is given as B
ANX	B	+ *	30h	48	Bitwise AND accumulator A with the contents of the location whose address is given as B, indexed by the value of X
ANI	B	+ *	31h	49	Bitwise AND accumulator A with the immediate value B
ORA	B	+ *	32h	50	Bitwise OR accumulator A with the contents of the location whose address is given as B
ORX	B	+ *	33h	51	Bitwise OR accumulator A with the contents of the location whose address is given as B, indexed by the value of X
ORI	B	+ *	34h	52	Bitwise OR accumulator A with the immediate value B
BRN	B		35h	53	Branch to the address given as B
BZE	B		36h	54	Branch to the address given as B if the Z condition flag is set
BNZ	B		37h	55	Branch to the address given as B if the Z condition flag is unset
BPZ	B		38h	56	Branch to the address given as B if the P condition flag is set
BNG	B		39h	57	Branch to the address given as B if the P condition flag is unset
BCC	B		3Ah	58	Branch to the address given as B if the C condition flag is unset
BCS	B		3Bh	59	Branch to the address given as B if the C condition flag is set
JSR	B		3Ch	60	Call subroutine whose address is B, pushing return address onto the stack

Most of the operations listed above are typical of those found in real machines. Notable exceptions are provided by the I/O (input/output) operations. Most real machines have extremely primitive facilities for doing anything like this directly, but for the purposes of this discussion we shall cheat somewhat and assume that our machine has several very powerful single-byte opcodes for handling I/O. (Actually this is not cheating too much, for some macro-assemblers allow instructions like this which are converted into procedure calls into part of an underlying operating system, stored perhaps in a ROM BIOS).

A careful examination of the machine and its instruction set will show some features that *are* typical of real machines. Although there are three data registers, `A, X` and `SP`, two of them (`X` and `SP`) can only be used in very specialized ways. For example, it is possible to transfer a value from `A` to `X`, but not vice versa, and while it is possible to load a value into `SP` it is not possible to examine the value of `SP` at a later stage. The logical operations affect the carry bit (they all unset it), but, surprisingly, the `INC` and `DEC` operations do not.

It is this model upon which we shall build an emulator in section 4.3.4. In a sense the formal semantics of these opcodes are then embodied directly in the **operational semantics** of the machine (or pseudo-machine) responsible for executing them.

Exercises

4.1 Which addressing mode is used in each of the operations defined above? Which addressing modes are not represented?

4.2 Many 8-bit microprocessors have 2-byte (16-bit) index registers, and one, two, and three-byte instructions (and even longer). What peculiar or restrictive features does our machine possess, compared to such processors?

4.3 As we have already commented, informal descriptions in English, as we have above, are not as precise as semantics that are formulated mathematically. Compare the informal description of the `INC` operation with the following:

```
INC  *  05h   5   A := (A + 1) mod 256;  Z := A = 0;  P := A IN {0 ... 127}
```

Try to express the semantics of each of the other machine instructions in a similar way.

4.3.3 A specimen program

Some examples of code for this machine may help the reader's understanding. Consider the problem of reading a number and then counting the number of non-zero bits in its binary representation.

Example 4.1

The listing below shows a program to solve this problem coded in an ASSEMBLER language based on the mnemonics given previously, as it might be listed by an assembler program, showing the hexadecimal representation of each byte and where it is located in memory.

```
00                  BEG                 ; Count the bits in a number
00    0A            INI                 ; Read(A)
01          LOOP                        ; REPEAT
01    16            SHR                 ;  A := A DIV 2
02    3A 0D         BCC     EVEN        ;  IF A MOD 2 # 0 THEN
04    1E 13         STA     TEMP        ;    TEMP := A
06    19 14         LDA     BITS
08    05            INC
09    1E 14         STA     BITS        ;    BITS := BITS + 1
0B    19 13         LDA     TEMP        ;    A := TEMP
0D    37 01 EVEN    BNZ     LOOP        ; UNTIL A = 0
0F    19 14         LDA     BITS        ;
11    0E            OTI                 ; Write(BITS)
12    18            HLT                 ; terminate execution
13          TEMP    DS      1           ; VAR TEMP : BYTE
14    00    BITS    DC      0           ;     BITS : BYTE
```

```
15                  END
```

Example 4.2 (absolute byte values)

In a later chapter we shall discuss how this same program can be translated into the following corresponding absolute format (expressed this time as decimal numbers):

```
10 22 58 13 30 19 25 20  5 30 20 25 19 55  1 25 20 14 24  0  0
```

Example 4.3 (mnemonics with absolute address fields)

For the moment, we shall allow ourselves to consider the absolute form as equivalent to a form in which the mnemonics still appear for the sake of clarity, but where the operands have all been converted into absolute (decimal) addresses and values:

```
INI
SHR
BCC   13
STA   19
LDA   20
INC
STA   20
LDA   19
BNZ   1
LDA   20
OTI
HLT
0
0
```

Exercises

4.4 The machine does not possess an instruction for negating the value in the accumulator. What code would one have to write to be able to achieve this?

4.5 Similarly, it does not possess instructions for multiplication and division. Is it possible to use the existing instructions to develop code for doing these operations? If so, how efficiently can they be done?

4.6 Try to write programs for this machine that will

(a) Find the largest of three numbers.

(b) Find the largest and the smallest of a list of numbers terminated by a zero (which is not regarded as a member of the list).

(c) Find the average of a list of non-zero numbers, the list being terminated by a zero.

(d) Compute N! for small N. Try using an iterative as well as a recursive approach.

(e) Read a word and then write it backwards. The word is terminated with a period. Try using an "array", or alternatively, the "stack".

(f) Determine the prime numbers between 0 and 255.

(g) Determine the longest repeated sequence in a sequence of digits terminated with

zero. For example, for data reading 1 2 3 3 3 3 4 5 4 4 4 4 4 4 4 6 5 5 report that "4 appeared 7 times".

(h) Read an input sequence of numbers terminated with zero, and then extract the embedded monotonically increasing sequence. For example, from 1 2 12 7 4 14 6 23 extract the sequence 1 2 12 14 23.

(i) Read a small array of integers or characters and sort them into order.

(j) Search for and report on the largest byte in the program code itself.

(k) Search for and report on the largest byte currently in memory.

(l) Read a piece of text terminated with a period, and then report on how many times each letter appeared. To make things interesting, ignore the difference between upper and lower case.

(m) Repeat some of the above problems using 16-bit arithmetic (storing values as pairs of bytes, and using the "carry" operations to perform extended arithmetic).

4.7 Based on your experiences with Exercise 4.6, comment on the usefulness, redundancy and any other features of the code set for the machine.

4.3.4 An emulator for the single-accumulator machine

Although a processor for our machine almost certainly does not exist "in silicon", its action may easily be simulated "in software". Essentially we need only to write an emulator that models the *fetch-execute* cycle of the machine, and we can do this in any suitable language for which we already have a compiler on a real machine.

Languages like Modula-2 or C++ are highly suited to this purpose. Not only do they have "bit-twiddling" capabilities for performing operations like "bitwise and", they have the advantage that one can implement the various phases of translators and emulators as coherent, clearly separated modules (in Modula-2) or classes (in C++). Extended versions of Pascal, such as Turbo Pascal, also provide support for such modules in the form of units. C is also very suitable on the first score, but is less well equipped to deal with clearly separated modules, as the header file mechanism used in C is less watertight than the mechanisms in the other languages.

In modelling our hypothetical machine in Modula-2 or C++ it will thus be convenient to define an interface in the usual way by means of a definition module, or by the public interface to a class. (In this text we shall illustrate code in C++; equivalent code in Modula-2 and Turbo Pascal will be found on the diskette that accompanies the book.)

The main responsibility of the interface is to declare an `emulator` routine for interpreting the code stored in the memory of the machine. For expediency we choose to extend the interface to expose the values of the operations, and the memory itself, and to provide various other useful facilities that will help us develop an assembler or compiler for the machine in due course. (In this, and in other interfaces, "private" members are not shown.)

```
// machine instructions - order is significant
enum MC_opcodes {
  MC_nop, MC_cla, MC_clc, MC_clx, MC_cmc, MC_inc, MC_dec, MC_inx, MC_dex,
  MC_tax, MC_ini, MC_inh, MC_inb, MC_ina, MC_oti, MC_otc, MC_oth, MC_otb,
```

```
    MC_ota, MC_psh, MC_pop, MC_shl, MC_shr, MC_ret, MC_hlt, MC_lda, MC_ldx,
    MC_ldi, MC_lsp, MC_lsi, MC_sta, MC_stx, MC_add, MC_adx, MC_adi, MC_adc,
    MC_acx, MC_aci, MC_sub, MC_sbx, MC_sbi, MC_sbc, MC_scx, MC_sci, MC_cmp,
    MC_cpx, MC_cpi, MC_ana, MC_anx, MC_ani, MC_ora, MC_orx, MC_ori, MC_brn,
    MC_bze, MC_bnz, MC_bpz, MC_bng, MC_bcc, MC_bcs, MC_jsr, MC_bad = 255 };

  typedef enum { running, finished, nodata, baddata, badop } status;
  typedef unsigned char MC_bytes;

  class MC {
    public:
      MC_bytes mem[256];    // virtual machine memory

      void listcode(void);
      // Lists the 256 bytes stored in mem on requested output file

      void emulator(MC_bytes initpc, FILE *data, FILE *results, bool tracing);
      // Emulates action of the instructions stored in mem, with program counter
      // initialized to initpc.  data and results are used for I/O.
      // Tracing at the code level may be requested

      void interpret(void);
      // Interactively opens data and results files, and requests entry point.
      // Then interprets instructions stored in mem

      MC_bytes opcode(char *str);
      // Maps str to opcode, or to MC_bad (0FFH) if no match can be found

      MC();
      // Initializes accumulator machine

  };
```

The implementation of `emulator` must model the typical *fetch-execute* cycle of the hypothetical machine. This is easily achieved by the repetitive execution of a large `switch` or `CASE` statement, and follows the lines of the algorithm given in section 4.1, but allowing for the possibility that the program may halt, or otherwise come to grief:

```
BEGIN
  InitializeProgramCounter(CPU.PC);
  InitializeRegisters(CPU.A, CPU.X, CPU.SP, CPU.Z, CPU.P, CPU.C);
  PS := running;
  REPEAT
    CPU.IR := Mem[CPU.PC]; Increment(CPU.PC)    (* fetch *)
    CASE CPU.IR OF                              (* execute *)
        . . . .
    END
  UNTIL PS # running;
  IF PS # finished THEN PostMortem END
END
```

A detailed implementation of the machine class is given as part of Appendix D, and the reader is urged to study it carefully.

Exercises

4.8 You will notice that the code in Appendix D makes no use of an explicit `EAR` register. Develop an emulator that does have such a register, and investigate whether this is an improvement.

4.9 How well does the informal description of the machine instruction set allow you to develop programs and an interpreter for the machine? Would a description in the form suggested by Exercise 4.3 be better?

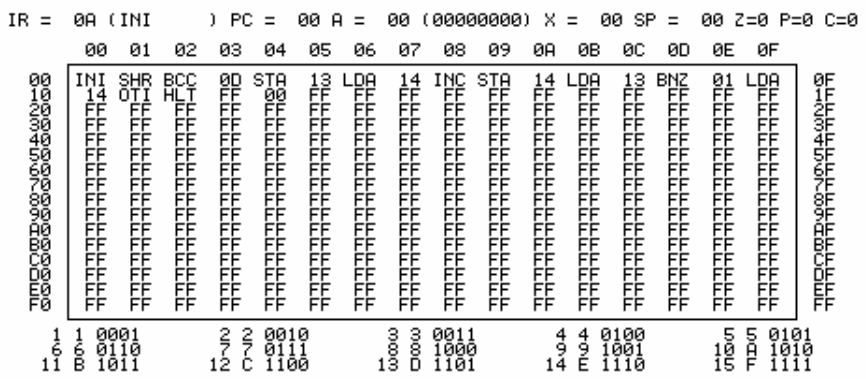
4.10 Do you suppose interpreters might find it difficult to handle I/O errors in user programs?

4.11 Although we have required that the machine incorporate the three condition flags `P, Z` and `C`, we have not provided another one commonly found on such machines, namely for detecting

overflow. Introduce V as such a flag into the definition of the machine, provide suitable instructions for testing it, and modify the emulator so that V is set and cleared by the appropriate operations.

4.12 Extend the instruction set and the emulator to include operations for negating the accumulator, and for providing multiplication and division operations.

4.13 Enhance the emulator so that when it interprets a program, a full screen display is given, highlighting the instruction that is currently being obeyed and depicting the entire memory contents of the machine, as well as the state of the machine registers. For example we might have a display like that in Figure 4.3 for the program exemplified earlier, at the stage where it is about to execute the first instruction.


IR = 0A (INI) PC = 00 A = 00 (00000000) X = 00 SP = 00 Z=0 P=0 C=0
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 INI SHR BCC 0D STA 13 LDA 14 INC STA 14 LDA 13 BNZ 01 LDA 0F
10 14 OTI HLT FF 00 FF FF FF FF FF FF FF FF FF FF FF 1F
20 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 2F
30 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 3F
40 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 4F
50 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 5F
60 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 6F
70 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 7F
80 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 8F
90 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 9F
A0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF AF
B0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF BF
C0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF CF
D0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF DF
E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF EF
F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
1 1 0001 2 2 0010 3 3 0011 4 4 0100 5 5 0101
6 6 0110 7 7 0111 8 8 1000 9 9 1001 10 A 1010
11 B 1011 12 C 1100 13 D 1101 14 E 1110 15 F 1111
Figure 4.3 Possible display format for an enhanced interpreter

4.3.5 A minimal assembler for the machine

Given the emulator as implemented above, and some way of assembling or compiling programs, it becomes possible to implement a complete load-and-go system for developing and running simple programs. An assembler can be provided through a class with a public interface like

```
class AS {
  public:
    AS(char *sourcename, MC *M);
    // Opens source file from supplied sourcename

    ~AS();
    // Closes source file

    void assemble(bool &errors);
    // Assembles source code from src file and loads bytes of code directly
    // into memory.  Returns errors = true if source code is corrupt
};
```

In terms of these two classes, a load-and-go system might then take the form

```
void main(int argc, char *argv[])
{ bool errors;
  if (argc == 1) { printf("Usage: ASSEMBLE source\n"); exit(1); }
  MC *Machine = new MC();
  AS *Assembler = new AS(argv[1], Machine);
  Assembler->assemble(errors);
  delete Assembler;
  if (errors)
    printf("Unable to interpret code\n");
  else
  { printf("Interpreting code ...\n");
    Machine->interpret();
  }
  delete Machine;
}
```

A detailed discussion of assembler techniques is given in a later chapter. For the moment we note that various implementations matching this interface might be written, of various complexities. The very simplest of these might require the user to hand-assemble his or her programs and would amount to nothing more than a simple loader:

```
AS::AS(char *sourcename, MC *M)
{ Machine = M;
  src = fopen(sourcename, "r");
  if (src == NULL) { printf("Could not open input file\n"); exit(1); }
}

AS::~AS()
{ if (src) fclose(src); src = NULL; }

void AS::assemble(bool &errors)
{ int number;
  errors = false;
  for (int i = 0; i <= 255; i++)
  { if (fscanf(src, "%d", &number) != 1)
      { errors = true; number = MC_bad; }
    Machine->mem[i] = number % 256;
  }
}
```

However, it is not difficult to write an alternative implementation of the `assemble` routine that allows the system to accept a sequence of mnemonics and numerical address fields, like that given in Example 4.3 earlier. We present possible code, with sufficient commentary that the reader should be able to follow it easily.

```
void readmnemonic(FILE *src, char &ch, char *mnemonic)
{ int i = 0;
  while (ch > ' ')
  { if (i <= 2) { mnemonic[i] = ch; i++; }
    ch = toupper(getc(src));
  }
  mnemonic[i] = '\0';
}

void readint(FILE *src, char &ch, int &number, bool &okay)
{ okay = true;
  number = 0;
  bool negative = (ch == '-');
  if (ch == '-' || ch == '+') ch = getc(src);
  while (ch > ' ')
  { if (isdigit(ch))
      number = number * 10 + ch - '0';
    else
      okay = false;
    ch = getc(src);
  }
  if (negative) number = -number;
}

void AS::assemble(bool &errors)
{ char mnemonic[4]; // mnemonic for matching
  MC_bytes lc = 0;  // location counter
  MC_bytes op;      // assembled opcode
  int number;       // assembled number
  char ch;          // general character for input
  bool okay;        // error checking on reading numbers

  printf("Assembling code ... \n");
  for (int i = 0; i <= 255; i++)          // fill with invalid opcodes
    Machine->mem[i] = MC_bad;
  lc = 0;                                 // initialize location counter
  errors = false;                         // optimist!
  do
  { do ch = toupper(getc(src));
    while (ch <= ' ' && !feof(src));      // skip spaces and blank lines
    if (!feof(src))                       // there should be a line to assemble
    { if (isupper(ch))                    // we should have a mnemonic
      { readmnemonic(src, ch, mnemonic);  // unpack it
        op = Machine->opcode(mnemonic);   // look it up
        if (op == MC_bad)                 // the opcode was unrecognizable
          { printf("%s - Bad mnemonic at %d\n", mnemonic, lc); errors = true; }
        Machine->mem[lc] = op;            // store numerical equivalent
```

```
      }
      else                              // we should have a numeric constant
      { readint(src, ch, number, okay);   // unpack it
        if (!okay) { printf("Bad number at %d\n", lc); errors = true; }
        if (number >= 0)                  // convert to proper byte value
          Machine->mem[lc] = number % 256;
        else
          Machine->mem[lc] = (256 - abs(number) % 256) % 256;
      }
      lc = (lc + 1) % 256;              // bump up location counter
    }
  } while (!feof(src));
}
```

4.4 Case study 2 - a stack-oriented computer

In later sections of this text we shall be looking at developing a compiler that generates object code for a hypothetical "stack machine", one that may have no general data registers of the sort discussed previously, but which functions primarily by manipulating a stack pointer and associated stack. An architecture like this will be found to be ideally suited to the evaluation of complicated arithmetic or Boolean expressions, as well as to the implementation of high-level languages which support recursion. It will be appropriate to discuss such a machine in the same way as we did for the single-accumulator machine in the last section.

4.4.1 Machine architecture

Compared with normal register based machines, this one may at first seem a little strange, because of the paucity of registers. In common with most machines we shall still assume that it stores code and data in a memory that can be modelled as a linear array. The elements of the memory are "words", each of which can store a single integer - typically using a 16 bit two’s-complement representation. Diagrammatically we might represent this machine as in Figure 4.4:

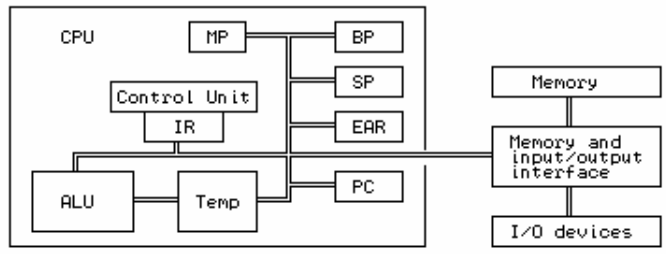


Figure 4.4 A simple stack-oriented CPU and computer

The symbols in this diagram refer to the following components of the machine

ALU is the *arithmetic logic unit* where arithmetic and logical operations are actually performed.

Temp is a set of 16-bit registers for holding intermediate results needed during arithmetic or logical operations. These registers cannot be accessed explicitly.

SP is the 16-bit *stack pointer*, a register that points to the area in memory utilized as the main stack.

BP is the 16-bit *base pointer*, a register that points to the base of an area of memory

within the stack, known as a *stack frame*, which is used to store variables.

`MP` is the 16-bit *mark stack pointer*, a register used in handling procedure calls, whose use will become apparent only in later chapters.

`IR` is the 16-bit *instruction register*, in which is held the instruction currently being executed.

`PC` is the 16-bit *program counter*, which contains the address in memory of the instruction that is the next to be executed.

`EAR` is the *effective address* register, which contains the address in memory of the data that is being manipulated by the current instruction.

A programmer's model of the machine is suggested by declarations like

```
CONST
  MemSize = 512;
TYPE
  ADDRESS = CARDINAL [0 .. MemSize - 1];
  PROCESSOR = RECORD
    IR : OPCODES;
    BP, MP, SP, PC : ADDRESS;
  END;
TYPE STATUS = (running, finished,
               badMem, badData,
               noData, divZero,
               badOP);
VAR
  CPU : PROCESSOR;
  Mem : ARRAY ADDRESS OF INTEGER;
  PS : STATUS;
```

```
const int MemSize = 512;
typedef short address;
struct processor {
  opcodes ir;
  address bp, mp, sp, pc;
};

typedef enum { running, finished,
  badmem, baddata, nodata,
  divzero, badop
} status;

processor cpu;
int mem[MemSize];
status ps;
```

For simplicity we shall assume that the code is stored in the low end of memory, and that the top part of memory is used as the stack for storing data. We shall assume that the topmost section of this stack is a *literal pool*, in which are stored constants, such as literal character strings. Immediately below this pool is the *stack frame*, in which the static variables are stored. The rest of the stack is to be used for working storage. A typical memory layout might be as shown in Figure 4.5, where the markers `CodeTop` and `StkTop` will be useful for providing memory protection in an emulated system.

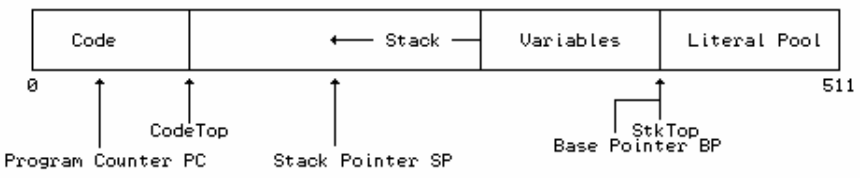


Figure 4.5 Usage of memory in the simple stack-oriented computer

We assume that the program loader will load the code at the bottom of memory (leaving the marker denoted by `CodeTop` pointing to the last word of code). It will also load the literals into the literal pool (leaving the marker denoted by `StkTop` pointing to the low end of this pool). It will go on to initialize both the stack pointer `SP` and base pointer `BP` to the value of `StkTop`. The first instruction in any program will have the responsibility of reserving further space on the stack for its variables, simply by decrementing the stack pointer `SP` by the number of words needed for these variables. A variable can be addressed by adding an offset to the base register `BP`. Since the stack "grows downwards" in memory, from high addresses towards low ones, these offsets will usually have

negative values.

4.4.2 Instruction set

A minimal set of operations for this machine is described informally below; in later chapters we shall find it convenient to add more opcodes to this set. We shall use the mnemonics introduced here to code programs for the machine in what appears to be a simple assembler language, albeit with addresses stipulated in absolute form.

Several of these operations belong to a category known as **zero address** instructions. Even though operands are clearly needed for operations such as addition and multiplication, the addresses of these are not specified by part of the instruction, but are implicitly derived from the value of the stack pointer `SP`. The two operands are assumed to reside on the top of the stack and just below the top; in our informal descriptions their values are denoted by `TOS` (for "top of stack") and `SOS` (for "second on stack"). A binary operation is performed by popping its two operands from the stack into (inaccessible) internal registers in the CPU, performing the operation, and then pushing the result back onto the stack. Such operations can be very economically encoded in terms of the storage taken up by the program code itself - the high density of stack-oriented machine code is another point in its favour so far as developing interpretive translators is concerned.

`ADD`		Pop `TOS` and `SOS`, add `SOS` to `TOS`, push sum to form new `TOS`
`SUB`		Pop `TOS` and `SOS`, subtract `TOS` from `SOS`, push result to form new `TOS`
`MUL`		Pop `TOS` and `SOS`, multiply `SOS` by `TOS`, push result to form new `TOS`
`DVD`		Pop `TOS` and `SOS`, divide `SOS` by `TOS`, push result to form new `TOS`
`EQL`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` = `TOS`, 0 otherwise
`NEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` # `TOS`, 0 otherwise
`GTR`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` > `TOS`, 0 otherwise
`LSS`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` < `TOS`, 0 otherwise
`LEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` <= `TOS`, 0 otherwise
`GEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` >= `TOS`, 0 otherwise
`NEG`		Negate `TOS`
`STK`		Dump stack to output (useful for debugging)
`PRN`		Pop `TOS` and write it to the output as an integer value
`PRS`	`A`	Write the nul-terminated string that was stacked in the literal pool from `Mem[A]`
`NLN`		Write a newline (carriage-return-line-feed) sequence
`INN`		Read integer value, pop `TOS`, store the value that was read in `Mem[TOS]`
`DSP`	`A`	Decrement value of stack pointer `SP` by `A`
`LIT`	`A`	Push the integer value `A` onto the stack to form new `TOS`
`ADR`	`A`	Push the value `BP + A` onto the stack to form new `TOS`. (This value is conceptually the address of a variable stored at an offset `A` within the stack frame pointed to by the base register `BP`.)
`IND`		Pop `TOS` to yield `Size`; pop `TOS` and `SOS`; if `0` <= `TOS` < `Size` then subtract `TOS` from `SOS`, push result to form new `TOS`
`VAL`		Pop `TOS`, and push the value of `Mem[TOS]` to form new `TOS` (an operation we shall call *dereferencing*)
`STO`		Pop `TOS` and `SOS`; store `TOS` in `Mem[SOS]`
`HLT`		Halt
`BRN`	`A`	Unconditional branch to instruction `A`
`BZE`	`A`	Pop `TOS`, and branch to instruction `A` if `TOS` is zero
`NOP`		No operation

The instructions in the first group are concerned with arithmetic and logical operations, those in the second group afford I/O facilities, those in the third group allow for the access of data in memory by means of manipulating addresses and the stack, and those in the last group allow for control of flow of the program itself. The `IND` operation allows for array indexing with subscript range

checking.

As before, the I/O operations are not typical of real machines, but will allow us to focus on the principles of emulation without getting lost in the trivia and overheads of handling real I/O systems.

Exercises

4.14 How closely does the machine code for this stack machine resemble anything you have seen before?

4.15 Notice that there is a `BZE` operation, but not a complementary `BNZ` (one that would branch if `TOS` were non-zero). Do you suppose this is a serious omission? Are there any opcodes which have been omitted from the set above which you can foresee as being absolutely essential (or at least very useful) for defining a viable "integer" machine?

4.16 Attempt to write down a mathematically oriented version of the semantics of each of the machine instructions, as suggested by Exercise 4.3.

4.4.3 Specimen programs

As before, some samples of program code for the machine may help to clarify various points.

Example 4.4

To illustrate how the memory is allocated, consider a simple section of program that corresponds to high-level code of the form

```
X := 8; Write("Y = ", Y);
```

```
                    ; Example 4.4
 0 DSP    2         ; X is at Mem[CPU.BP-1], Y is at Mem[CPU.BP-2]
 2 ADR   -1         ; push address of X
 4 LIT    8         ; push 8
 6 STO              ;        X := 8
 7 STK              ; dump stack to look at it
 8 PRS   'Y = '     ;        Write string "Y = "
10 ADR   -2         ; push address of Y
12 VAL              ; dereference
13 PRN              ;        Write integer Y
14 HLT              ; terminate execution
```

This would be stored in memory as

```
 DSP  2  ADR -1  LIT  8  STO STK PRS 510 ADR -2  VAL PRN HLT
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14

 ...   (Y)   (X)   0   ' '  '='  ' '  'Y'   0
       504   505  506  507  508  509  510  511
```

Immediately after loading this program (and before executing the `DSP` instruction), the program counter `PC` would have the value 0, while the base register `BP` and stack pointer `SP` would each have the value 506.

Example 4.5

Example 4.4 scarcely represents the epitome of the programmer's art! A more ambitious program follows, as a translation of the simple algorithm

```
BEGIN
  Y := 0;
  REPEAT  READ(X);  Y := X + Y   UNTIL X = 0;
  WRITE('Total is ', Y);
END
```

This would require a stack frame of size two to contain the variables X and Y. The machine code might read

```
                    ;  Example 4.5
 0 DSP    2         ;  X is at Mem[CPU.BP-1], Y is at Mem[CPU.BP-2]
 2 ADR   -2         ;  push address of Y (CPU.BP-2) on stack
 4 LIT    0         ;  push 0 on stack
 6 STO              ;  store 0 as value of Y
 7 ADR   -1         ;  push address of X (CPU.BP-1) on stack
 9 INN              ;  read value, store on X
10 ADR   -2         ;  push address of Y on stack
12 ADR   -1         ;  push address of X on stack
14 VAL              ;  dereference - value of X now on stack
15 ADR   -2         ;  push address of Y on stack
17 VAL              ;  dereference - value of Y now on stack
18 ADD              ;  add X to Y
19 STO              ;  store result as new value of Y
20 ADR   -1         ;  push address of X on stack
22 VAL              ;  dereference - value of X now on stack
23 LIT    0         ;  push constant 0 onto stack
25 EQL              ;  check equality
26 BZE    7         ;  branch if X # 0
28 PRS   'Total is' ;  label output
30 ADR   -2         ;  push address of Y on stack
32 VAL              ;  dereference - value of Y now on stack
33 PRN              ;  write result
34 HLT              ;  terminate execution
```

Exercises

4.17 Would you write code anything like that given in Example 4.5 if you had to translate the corresponding algorithm into a familiar ASSEMBLER language directly?

4.18 How difficult would it be to hand translate programs written in this stack machine code into your favourite ASSEMBLER ?

4.19 Use the stack language (and, in due course, its interpreter) to write and test the simple programs suggested in Exercises 4.6.

4.4.4 An emulator for the stack machine

Once again, to emulate this machine by means of a program written in Modula-2 or C++, it will be convenient to define an interface to the machine by means of a definition module or appropriate class. As in the case of the accumulator machine, the main exported facility is a routine to perform the emulation itself, but for expediency we shall export further entities that make it easy to develop an assembler, compiler, or loader that will leave pseudo-code directly in memory after translation of some source code.

```
const int STKMC_memsize = 512;  // Limit on memory

// machine instructions - order is significant
enum STKMC_opcodes {
  STKMC_adr, STKMC_lit, STKMC_dsp, STKMC_brn, STKMC_bze, STKMC_prs, STKMC_add,
  STKMC_sub, STKMC_mul, STKMC_dvd, STKMC_eql, STKMC_neq, STKMC_lss, STKMC_geq,
  STKMC_gtr, STKMC_leq, STKMC_neg, STKMC_val, STKMC_sto, STKMC_ind, STKMC_stk,
  STKMC_hlt, STKMC_inn, STKMC_prn, STKMC_nln, STKMC_nop, STKMC_nul
};
```

```
typedef enum {
  running, finished, badmem, baddata, nodata, divzero, badop, badind
} status;
typedef int STKMC_address;

class STKMC {
  public:
    int mem[STKMC_memsize];  // virtual machine memory

    void listcode(char *filename, STKMC_address codelen);
    // Lists the codelen instructions stored in mem on named output file

    void emulator(STKMC_address initpc, STKMC_address codelen,
                  STKMC_address initsp, FILE *data, FILE *results,
                  bool tracing);
    // Emulates action of the codelen instructions stored in mem, with
    // program counter initialized to initpc, stack pointer initialized to
    // initsp.  data and results are used for I/O.  Tracing at the code level
    // may be requested

    void interpret(STKMC_address codelen, STKMC_address initsp);
    // Interactively opens data and results files.  Then interprets the
    // codelen instructions stored in mem, with stack pointer initialized
    // to initsp

    STKMC_opcodes opcode(char *str);
    // Maps str to opcode, or to STKMC_nul if no match can be found

    STKMC();
    // Initializes stack machine
};
```

The emulator itself has to model the typical *fetch-execute* cycle of an actual machine. This is easily achieved as before, and follows an almost identical pattern to that used for the other machine. A full implementation is to be found on the accompanying diskette; only the important parts are listed here for the reader to study:

```
bool STKMC::inbounds(int p)
// Check that memory pointer p does not go out of bounds.  This should not
// happen with correct code, but it is just as well to check
{ if (p < stackmin || p >= STKMC_memsize) ps = badmem;
  return (ps == running);
}

void STKMC::stackdump(STKMC_address initsp, FILE *results, STKMC_address pcnow)
// Dump data area - useful for debugging
{ int online = 0;
  fprintf(results, "\nStack dump at %4d", pcnow);
  fprintf(results, " SP:%4d BP:%4d SM:%4d\n", cpu.sp, cpu.bp, stackmin);
  for (int l = stackmax - 1; l >= cpu.sp; l--)
  { fprintf(results, "%7d:%5d", l, mem[l]);
    online++; if (online % 6 == 0) putc('\n', results);
  }
  putc('\n', results);
}

void STKMC::trace(FILE *results, STKMC_address pcnow)
// Simple trace facility for run time debugging
{ fprintf(results, " PC:%4d BP:%4d SP:%4d TOS:", pcnow, cpu.bp, cpu.sp);
  if (cpu.sp < STKMC_memsize)
    fprintf(results, "%4d", mem[cpu.sp]);
  else
    fprintf(results, "????");
  fprintf(results, " %s", mnemonics[cpu.ir]);
  switch (cpu.ir)
  { case STKMC_adr:
    case STKMC_prs:
    case STKMC_lit:
    case STKMC_dsp:
    case STKMC_brn:
    case STKMC_bze:
      fprintf(results, "%7d", mem[cpu.pc]); break;
    // no default needed
  }
  putc('\n', results);
}

void STKMC::postmortem(FILE *results, STKMC_address pcnow)
// Report run time error and position
{ putc('\n', results);
```

```
  switch (ps)
  { case badop:    fprintf(results, "Illegal opcode"); break;
    case nodata:   fprintf(results, "No more data"); break;
    case baddata:  fprintf(results, "Invalid data"); break;
    case divzero:  fprintf(results, "Division by zero"); break;
    case badmem:   fprintf(results, "Memory violation"); break;
    case badind:   fprintf(results, "Subscript out of range"); break;
  }
  fprintf(results, " at %4d\n", pcnow);
}

void STKMC::emulator(STKMC_address initpc, STKMC_address codelen,
                     STKMC_address initsp, FILE *data, FILE *results,
                     bool tracing)
{ STKMC_address pcnow;  // current program counter
  stackmax = initsp;
  stackmin = codelen;
  ps = running;
  cpu.sp = initsp;
  cpu.bp = initsp;      // initialize registers
  cpu.pc = initpc;      // initialize program counter
  do
  { pcnow = cpu.pc;
    if (unsigned(mem[cpu.pc]) > int(STKMC_nul)) ps = badop;
    else
    { cpu.ir = STKMC_opcodes(mem[cpu.pc]); cpu.pc++;  // fetch
      if (tracing) trace(results, pcnow);
      switch (cpu.ir)                                 // execute
      { case STKMC_adr:
          cpu.sp--;
          if (inbounds(cpu.sp))
            { mem[cpu.sp] = cpu.bp + mem[cpu.pc]; cpu.pc++; }
          break;
        case STKMC_lit:
          cpu.sp--;
          if (inbounds(cpu.sp)) { mem[cpu.sp] = mem[cpu.pc]; cpu.pc++; }
          break;
        case STKMC_dsp:
          cpu.sp -= mem[cpu.pc];
          if (inbounds(cpu.sp)) cpu.pc++;
          break;
        case STKMC_brn:
          cpu.pc = mem[cpu.pc]; break;
        case STKMC_bze:
          cpu.sp++;
          if (inbounds(cpu.sp))
          { if (mem[cpu.sp - 1] == 0) cpu.pc = mem[cpu.pc]; else cpu.pc++; }
          break;
        case STKMC_prs:
          if (tracing) fputs(BLANKS, results);
          int loop = mem[cpu.pc];
          cpu.pc++;
          while (inbounds(loop) && mem[loop] != 0)
            { putc(mem[loop], results); loop--; }
          if (tracing) putc('\n', results);
          break;
        case STKMC_add:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] += mem[cpu.sp - 1];
          break;
        case STKMC_sub:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
          break;
        case STKMC_mul:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] *= mem[cpu.sp - 1];
          break;
        case STKMC_dvd:
          cpu.sp++;
          if (inbounds(cpu.sp))
          { if (mem[cpu.sp - 1] == 0)
              ps = divzero;
            else
              mem[cpu.sp] /= mem[cpu.sp - 1];
          }
          break;
        case STKMC_eql:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] == mem[cpu.sp - 1]);
          break;
        case STKMC_neq:
          cpu.sp++;
```

```
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] != mem[cpu.sp - 1]);
          break;
        case STKMC_lss:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] < mem[cpu.sp - 1]);
          break;
        case STKMC_geq:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] >= mem[cpu.sp - 1]);
          break;
        case STKMC_gtr:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] > mem[cpu.sp - 1]);
          break;
        case STKMC_leq:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] <= mem[cpu.sp - 1]);
          break;
        case STKMC_neg:
          if (inbounds(cpu.sp)) mem[cpu.sp] = -mem[cpu.sp];
          break;
        case STKMC_val:
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[cpu.sp] = mem[mem[cpu.sp]];
          break;
        case STKMC_sto:
          cpu.sp++;
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[mem[cpu.sp]] = mem[cpu.sp - 1];
          cpu.sp++;
          break;
        case STKMC_ind:
          if ((mem[cpu.sp + 1] < 0) || (mem[cpu.sp + 1] >= mem[cpu.sp]))
            ps = badind;
          else
          { cpu.sp += 2;
            if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
          }
          break;
        case STKMC_stk:
          stackdump(initsp, results, pcnow); break;
        case STKMC_hlt:
          ps = finished; break;
        case STKMC_inn:
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
          { if (fscanf(data, "%d", &mem[mem[cpu.sp]]) == 0)
              ps = baddata;
            else
              cpu.sp++;
          }
          break;
        case STKMC_prn:
          if (tracing) fputs(BLANKS, results);
          cpu.sp++;
          if (inbounds(cpu.sp)) fprintf(results, " %d", mem[cpu.sp - 1]);
          if (tracing) putc('\n', results);
          break;
        case STKMC_nln:
          putc('\n', results); break;
        case STKMC_nop:
          break;
        default:
          ps = badop; break;
      }
    }
  } while (ps == running);
  if (ps != finished) postmortem(results, pcnow);
}
```

We should remark that there is rather more error-checking code in this interpreter than we should like. This will detract from the efficiency of the interpreter, but is code that is probably very necessary when testing the system.

Exercises

4.20 Can you think of ways in which this interpreter can be improved, both as regards efficiency, and user friendliness? In particular, try adding debugging aids over and above the simple stack dump already provided. Can you think of any ways in which it could be made to detect infinite loops in a user program, or to allow itself to be manually interrupted by an irate or frustrated user?

4.21 The interpreter attempts to prevent corruption of the memory by detecting when the machine registers go out of bounds. The implementation above is not totally foolproof so, as a useful exercise, improve on it. One might argue that correct code will never cause such corruption to occur, but if one attempts to write stack machine code by hand, it will be found easy to "push" without "popping" or *vice versa*, and so the checks are very necessary.

4.22 The interpreter checks for division by zero, but does no other checking that arithmetic operations will stay within bounds. Improve it so that it does so, bearing in mind that one has to predict overflow, rather than wait for it to occur.

4.23 As an alternative, extend the machine so that overflow detection does not halt the program, but sets an overflow flag in the processor. Provide operations whereby the programmer can check this flag and take whatever action he or she deems appropriate.

4.24 One of the advantages of an emulated machine is that it is usually very easy to extend it (provided the host language for the interpreter can support the features required). Try introducing two new operations, say `INC` and `PRC`, which will read and print single character data. Then rework those of Exercises 4.6 that involve characters.

4.25 If you examine the code in Examples 4.4 and 4.5 - and in the solutions to Exercises 4.6 - you will observe that the sequences

```
ADR x
VAL
```

and

```
ADR x
(calculations)
STO
```

are very common. Introduce and implement two new operations

```
PSH  A     Push   Mem[CPU.BP + A]   onto stack to form new  TOS
POP  A     Pop   TOS   and assign   Mem[CPU.BP + A] := TOS
```

Then rework some of Exercise 4.6 using these facilities, and comment on the possible advantages of having these new operations available.

4.26 As a further variation on the emulated machine, develop a variation where the branch instructions are "relative" rather than "absolute". This makes for rather simpler transition to relocatable code.

4.27 Is it possible to accomplish Boolean (NOT, AND and OR) operations using the current instruction set? If not, how would you extend the instruction set to incorporate these? If they are not strictly necessary, would they be useful additions anyway?

4.28 As yet another alternative, suppose the machine had a set of condition flags such as `Z` and `P`, similar to those used in the single-accumulator machine of the last section. How would the instruction set and the emulator need to be changed to use these? Would their presence make it

easier to write programs, particularly those that need to evaluate complex Boolean expressions?

4.4.5 A minimal assembler for the machine

To be able to use this system we must, of course, have some way of loading or assembling code into memory. An assembler might conveniently be developed using the following interface, very similar to that used for the single- accumulator machine.

```
class STKASM {
  public:
    STKASM(char *sourcename, STKMC *M);
    // Opens source file from supplied sourcename

    ~STKASM();
    // Closes source file

    void assemble(bool &errors, STKMC_address &codetop,
                  STKMC_address &stktop);
    // Assembles source code from an input file and loads codetop
    // words of code directly into memory mem[0 .. codetop-1],
    // storing strings in the string pool at the top of memory in
    // mem[stktop .. STKMC_memsize-1].
    //
    // Returns
    //    codetop = number of instructions assembled and stored
    //              in mem[0] .. mem[codetop - 1]
    //    stktop  = 1 + highest byte in memory available
    //              below string pool in mem[stktop] .. mem[STK_memsize-1]
    //    errors  = true if erroneous instruction format detected
    // Instruction format :
    //    Instruction  = [Label] Opcode [AddressField] [Comment]
    //    Label        = Integer
    //    Opcode       = STKMC_Mnemonic
    //    AddressField = Integer | 'String'
    //    Comment      = String
    //
    // A string AddressField may only be used with a PRS opcode
    // Instructions are supplied one to a line; terminated at end of input file
};
```

This interface would allow us to develop sophisticated assemblers without altering the rest of the system - merely the implementation. In particular we can write a load-and-go assembler/interpreter very easily, using essentially the same system as was suggested in section 4.3.5.

The objective of this chapter is to introduce the principles of machine emulation, and not to be too concerned about the problems of assembly. If, however, we confine ourselves to assembling code where the operations are denoted by their mnemonics, but all the addresses and offsets are written in absolute form, as was done for Examples 4.4 and 4.5, a rudimentary assembler can be written relatively easily. The essence of this is described informally by an algorithm like

```
BEGIN
  CodeTop := 0;
  REPEAT
    SkipLabel;
    IF NOT EOF(SourceFile) THEN
      Extract(Mnemonic);
      Convert(Mnemonic, OpCode);
      Mem[CodeTop] := OpCode; Increment(CodeTop);
      IF OpCode = PRS THEN
        Extract(String); Store(String, Address);
        Mem[CodeTop] := Address; Increment(CodeTop);
      ELSIF OpCode in {ADR, LIT, DSP, BRN, BZE} THEN
        Extract(Address); Mem[CodeTop] := Address; Increment(CodeTop);
      END;
      IgnoreComments;
    END
  UNTIL EOF(SourceFile)
END
```

An implementation of this is to be found on the source diskette, where code is assumed to be

supplied to the machine in free format, one instruction per line. Comments and labels may be added, as in the examples given earlier, but these are simply ignored by the assembler. Since absolute addresses are required, any labels are more of a nuisance than they are worth.

Exercises

4.29 The assembler on the source diskette attempts some, but not much, error detection. Investigate how it could be improved.

4.30 The machine is rather wasteful of memory. Had we used a byte oriented approach we could have stored the code and the literal strings far more compactly. Develop an implementation that does this.

4.31 It might be deemed unsatisfactory to locate the literal pool in high memory. An alternative arrangement would be to locate it immediately above the executable code, on the lines of Figure 4.6. Develop a variation on the assembler (and, if necessary, the interpreter) to exploit this idea.

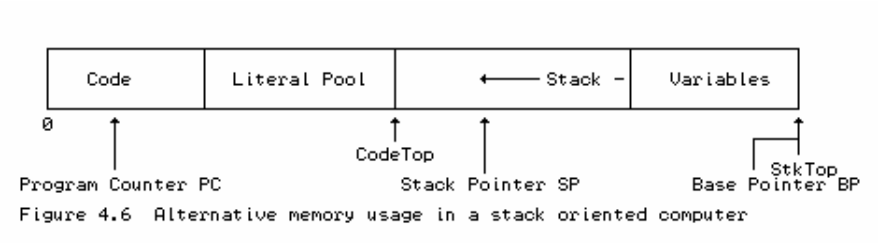


Figure 4.6 Alternative memory usage in a stack oriented computer

---

Further reading

Other descriptions of pseudo-machines and of stack machines are to be found in the books by Wakerly (1981), Brinch Hansen (1985), Wirth (1986, 1996), Watt (1993), and Bennett (1990).

The very comprehensive stack-based interpreter for the Zürich Pascal-P system is fully described in the book by Pemberton and Daniels (1982).

5 LANGUAGE SPECIFICATION

A study of the syntax and semantics of programming languages may be made at many levels, and is an important part of modern Computer Science. One can approach it from a very formal viewpoint, or from a very informal one. In this chapter we shall mainly be concerned with ways of specifying the concrete syntax of languages in general, and programming languages in particular. This forms a basis for the further development of the syntax- directed translation upon which much of the rest of this text depends.

5.1 Syntax, semantics, and pragmatics

People use languages in order to communicate. In ordinary speech they use natural languages like English or French; for more specialized applications they use technical languages like that of mathematics, for example

$$\forall x \exists \varepsilon :: | x - \xi | < \varepsilon$$

We are mainly concerned with programming languages, which are notations for describing computations. (As an aside, the word "language" is regarded by many to be unsuitable in this context. The word "notation" is preferable; we shall, however, continue to use the traditional terminology.) A useful programming language must be suited both to *describing* and to *implementing* the solution to a problem, and it is difficult to find languages which satisfy both requirements - efficient implementation seems to require the use of low-level languages, while easy description seems to require the use of high-level languages.

Most people are taught their first programming language by example. This is admirable in many respects, and probably unavoidable, since learning the language is often carried out in parallel with the more fundamental process of learning to develop algorithms. But the technique suffers from the drawback that the tuition is incomplete - after being shown only a limited number of examples, one is inevitably left with questions of the "can I do this?" or "how do I do this?" variety. In recent years a great deal of effort has been spent on formalizing programming (and other) languages, and in finding ways to describe them and to define them. Of course, a formal programming language has to be described by using another language. This language of description is called the **metalanguage**. Early programming languages were described using English as the metalanguage. A precise specification requires that the metalanguage be completely unambiguous, and this is not a strong feature of English (politicians and comedians rely heavily on ambiguity in spoken languages in pursuing their careers!). Some beginner programmers find that the best way to answer the questions which they have about a programming language is to ask them of the compilers which implement the language. This is highly unsatisfactory, as compilers are known to be error-prone, and to differ in the way they handle a particular language.

Natural languages, technical languages and programming languages are alike in several respects. In each case the **sentences** of a language are composed of sets of **strings** of **symbols** or **tokens** or **words**, and the construction of these sentences is governed by the application of two sets of rules.

- **Syntax Rules** describe the *form* of the sentences in the language. For example, in English, the

sentence "They can fish" is syntactically correct, while the sentence "Can fish they" is incorrect. To take another example, the language of binary numerals uses only the symbols 0 and 1, arranged in strings formed by concatenation, so that the sentence 101 is syntactically correct for this language, while the sentence 1110211 is syntactically incorrect.

- **Semantic Rules**, on the other hand, define the *meaning* of syntactically correct sentences in a language. By itself the sentence 101 has no meaning without the addition of semantic rules to the effect that it is to be interpreted as the representation of some number using a positional convention. The sentence "They can fish" is more interesting, for it can have two possible meanings; a set of semantic rules would be even harder to formulate.

The formal study of syntax as applied to programming languages took a great step forward in about 1960, with the publication of the *Algol 60 report* by Naur (1960, 1963), which used an elegant, yet simple, notation known as **Backus-Naur-Form** (sometimes called **Backus-Normal-Form**) which we shall study shortly. Simply understood notations for describing semantics have not been so forthcoming, and many semantic features of languages are still described informally, or by example.

Besides being aware of syntax and semantics, the user of a programming language cannot avoid coming to terms with some of the pragmatic issues involved with implementation techniques, programming methodology, and so on. These factors govern subtle aspects of the design of almost every practical language, often in a most irritating way. For example, in Fortran 66 and Fortran 77 the length of an identifier was restricted to a maximum of six characters - a legacy of the word size on the IBM computer for which the first Fortran compiler was written.

5.2 Languages, symbols, alphabets and strings

In trying to specify programming languages rigorously one must be aware of some features of **formal language theory**. We start with a few abstract definitions:

- A **symbol** or **token** is an atomic entity, represented by a character, or sometimes by a reserved or key word, for example `+ , ; END`.

- An **alphabet** *A* is a non-empty, but finite, set of symbols. For example, the alphabet of Modula-2 includes the symbols

  ```
  - / * a b c A B C BEGIN CASE END
  ```

  while that for C++ would include a corresponding set

  ```
  - / * a b c A B C { switch }
  ```

- A **phrase**, **word** or **string** "over" an alphabet *A* is a sequence $\sigma = a_1 a_2 ... a_n$ of symbols from *A*.

- It is often useful to hypothesize the existence of a string of length zero, called the **null string** or **empty word**, usually denoted by ε (some authors use λ instead). This has the property that if it is concatenated to the left or right of any word, that word remains unaltered.

  $$a \varepsilon = \varepsilon a = a$$

- The set of all strings of length n over an alphabet A is denoted by A^n. The set of all strings (including the null string) over an alphabet A is called its **Kleene closure** or, simply, **closure**, and is denoted by A^*. The set of all strings of length at least one over an alphabet A is called its **positive closure**, and is denoted by A^+. Thus

  $$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \ldots$$

- A **language** L over an alphabet A is a subset of A^*. At the present level of discussion this involves no concept of meaning. A language is simply a set of strings. A language consisting of a finite number of strings can be defined simply by listing all those strings, or giving a rule for their derivation. This may even be possible for simple infinite languages. For example, we might have

  $$L = \{ ([a+)^n (b])^n \mid n > 0 \}$$

  (the vertical stroke can be read "such that"), which defines exciting expressions like

  $[a + b]$
  $[a + [a + b]\ b]$
  $[a + [a + [a + b]\ b]\ b]$

5.3 Regular expressions

Several simple languages - but by no means all - can be conveniently specified using the notation of **regular expressions**. A regular expression specifies the form that a string may take by using the symbols from the alphabet A in conjunction with a few other **metasymbols**, which represent operations that allow for

- *Concatenation* - symbols or strings may be concatenated by writing them next to one another, or by using the metasymbol · (dot) if further clarity is required.

- *Alternation* - a choice between two symbols a and b is indicated by separating them by the metasymbol **|** (bar).

- *Repetition* - a symbol a followed by the metasymbol ***** (star) indicates that a sequence of zero or more occurrences of a is allowable.

- *Grouping* - a group of symbols may be surrounded by the metasymbols **(** and **)** (parentheses).

As an example of a regular expression, consider

$$1 (1 \mid 0)^* 0$$

This generates the set of strings, each of which has a leading 1, is followed by any number of 0’s or 1’s, and is terminated with a 0 - that is, the set

{ 10, 100, 110, 1000 ... }

If a semantic interpretation is required, the reader will recognize this as the set of strings representing non-zero even numbers in a binary representation,

Formally, regular expressions may be defined inductively as follows:

- A regular expression denotes a regular set of strings.
- Ø is a regular expression denoting the empty set.
- ε is a regular expression denoting the set that contains only the empty string.
- σ is a regular expression denoting a set containing only the string σ.
- If *A* and *B* are regular expressions, then (*A*) and *A* | *B* and *A* · *B* and A^* are also regular expressions.

Thus, for example, if σ and τ are strings generated by regular expressions, στ and σ · τ are also generated by a regular expression.

The reader should take note of the following points:

- As in arithmetic, where multiplication and division take precedence over addition and subtraction, there is a precedence ordering between these operators. Parentheses take precedence over repetition, which takes precedence over concatenation, which in turn takes precedence over alternation. Thus, for example, the following two regular expressions are equivalent

  his | hers and h (i | er) s

  and both define the set of strings { his , hers }.

- If the metasymbols are themselves allowed to be members of the alphabet, the convention is to enclose them in quotes when they appear as simple symbols within the regular expression. For example, comments in Pascal may be described by the regular expression

  "(" "*" c^* "*" ")" where $c \in A$

- Some other shorthand is commonly found. For example, the positive closure symbol $^+$ is sometimes used, so that a^+ is an alternative representation for $a\ a^*$. A question mark is sometimes used to denote an optional instance of *a*, so that *a*? denotes *a* | ε. Finally, brackets and hyphens are often used in place of parentheses and bars, so that [*a*-*eBC*] denotes (*a* | *b* | *c* | *d* | *e* | *B* | *C*).

- Regular expressions have a variety of algebraic properties, among which we can draw attention to

```
A | B = B | A                    (commutativity for alternation)
A | ( B | C ) = ( A | B ) | C    (associativity for alternation)
A | A  =  A                      (absorption for alternation)
A · ( B · C ) = ( A · B ) · C    (associativity for concatenation)
A ( B | C ) = A B | A C          (left distributivity)
( A | B ) C = A C | B C          (right distributivity)
A ε = ε A = A                    (identity for concatenation)
```

$A^* A^* = A^*$ (absorption for closure)

Regular expressions are of practical interest in programming language translation because they can be used to specify the structure of the tokens (like identifiers, literal constants, and comments) whose recognition is the prerogative of the scanner (lexical analyser) phase of a compiler.

For example, the set of integer literals in many programming languages is described by the regular expression

$$(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+$$

or, more verbosely, by

$$(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \cdot (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$$

or, more concisely, by

$$[0\text{-}9]^+$$

and the set of identifiers by a similar regular expression

$$(a \mid b \mid c \mid \ldots \mid Z) \cdot (0 \mid 1 \mid \ldots \mid 9 \mid a \mid \ldots \mid Z)^*$$

or, more concisely, by

$$[a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*$$

Regular expressions are also powerful enough to describe complete simple assembler languages of the forms illustrated in the last chapter, although the complete expression is rather tedious to write down, and so is left as an exercise for the zealous reader.

Exercises

5.1 How would the regular expression for even binary numbers need modification if the string 0 (zero) was allowed to be part of the language?

5.2 In some programming languages, identifiers may have embedded underscore characters. However, the first character may not be an underscore, nor may two underscores appear in succession. Write a regular expression that generates such identifiers.

5.3 Can you find regular expressions that describe the form of `REAL` literal constants in Pascal? In C++? In Modula-2?

5.4 Find a regular expression that generates the Roman representation of numbers from 1 through 99.

5.5 Find a regular expression that generates strings like "facetious" and "abstemious" that contain all five vowels, in order, but appearing only once each.

5.6 Find a regular expression that generates all strings of 0's and 1's that have an odd number of 0's and an even number of 1's.

5.7 Describe the simple assembler languages of the last chapter by means of regular expressions.

5.4 Grammars and productions

Most practical languages are, of course, rather more complicated than can be defined by regular expressions. In particular, regular expressions are not powerful enough to describe languages that manifest *self-embedding* in their descriptions. Self-embedding comes about, for example, in describing structured statements which have components that can themselves be statements, or expressions comprised of factors that may contain further parenthesized expressions, or variables declared in terms of types that are structured from other types, and so on.

Thus we move on to consider the notion of a **grammar**. This is essentially a set of rules for describing **sentences** - that is, choosing the subsets of A^* in which one is interested. Formally, a grammar G is a quadruple $\{ N, T, S, P \}$ with the four components

(a) N - a finite set of **non-terminal** symbols,
(b) T - a finite set of **terminal** symbols,
(c) S - a special **goal** or **start** or **distinguished** symbol,
(d) P - a finite set of **production rules** or, simply, **productions**.

(The word "set" is used here in the mathematical sense.) A sentence is a string composed entirely of terminal symbols chosen from the set T. On the other hand, the set N denotes the **syntactic classes** of the grammar, that is, general components or concepts used in describing sentence construction.

The union of the sets N and T denotes the **vocabulary** V of the grammar.

$$V = N \cup T$$

and the sets N and T are required to be disjoint, so that

$$N \cap T = \emptyset$$

where $\emptyset$ is the empty set.

A convention often used when describing grammars in the abstract is to use lower-case Greek letters (α, β, γ, ...) to represent strings of terminals and/or non-terminals, capital Roman letters (A, B, C ...) to represent single non- terminals and lower case Roman letters (a, b, c ...) to represent single terminals. Each author seems to have his or her own set of conventions, so the reader should be on guard when consulting the literature. Furthermore, when referring to the types of strings generated by productions, use is often made of the closure operators. Thus, if a string α consists of zero or more terminals (and no non-terminals) we should write

$$\alpha \in T^*$$

while if α consists of one or more non-terminals (but no terminals)

$\alpha \in N^+$

and if α consists of zero or more terminals and/or non-terminals

$\alpha \in (N \cup T)^*$ that is, $\alpha \in V^*$

English words used as the names of non-terminals, like *sentence* or *noun* are often non-terminals. When describing programming languages, reserved or key words (like `END, BEGIN` and `CASE`) are inevitably terminals. The distinction between these is sometimes made with the use of different type face - we shall use *italic font* for non-terminals and `monospaced font` for terminals where it is necessary to draw a firm distinction.

This probably all sounds very abstruse, so let us try to enlarge a little, by considering English as a written language. The set *T* here would be one containing the 26 letters of the common alphabet, and punctuation marks. The set *N* would be the set containing syntactic descriptors - simple ones like *noun*, *adjective*, *verb*, as well as more complex ones like *noun phrase*, *adverbial clause* and *complete sentence*. The set *P* would be one containing syntactic rules, such as a description of a *noun phrase* as a sequence of *adjective* followed by *noun*. Clearly this set can become very large indeed - much larger than *T* or even *N*. The productions, in effect, tell us how we can *derive* sentences in the language. We start from the distinguished symbol *S*, (which is always a non-terminal such as *complete sentence*) and, by making successive substitutions, work through a sequence of so- called **sentential forms** towards the final string, which contains terminals only.

There are various ways of specifying productions. Essentially a production is a rule relating to a pair of strings, say γ and δ, specifying how one may be transformed into the other. Sometimes they are called **rewrite rules** or **syntax equations** to emphasize this property. One way of denoting a general production is

$\gamma \rightarrow \delta$

To introduce our last abstract definitions, let us suppose that σ and τ are two strings each consisting of zero or more non-terminals and/or terminals (that is, $\sigma, \tau \in V = (N \cup T)^*$).

- If we can obtain the string τ from the string σ by employing *one* of the productions of the grammar *G*, then we say that σ *directly produces* τ (or that τ *is directly derived from* σ), and express this as $\sigma \Rightarrow \tau$.

  That is, if $\sigma = \alpha\delta\beta$ and $\tau = \alpha\gamma\beta$, and $\delta \rightarrow \gamma$ is a production in *G*, then $\sigma \Rightarrow \tau$.

- If we can obtain the string τ from the string σ by applying *n* productions of *G*, with $n \geq 1$, then we say that σ *produces* τ *in a non-trivial way* (or that τ *is derived from* σ *in a non-trivial way*), and express this as $\sigma \Rightarrow^+ \tau$.

  That is, if there exists a sequence $\alpha_o, \alpha_1, \alpha_2, \ldots \alpha_k$ (with $k \geq 1$), such that

  $\sigma = \alpha_o$,
  $\alpha_{j-1} \Rightarrow \alpha_j$ (for $1 \leq j \leq k$)
  $\alpha_k = \tau$,

then $\sigma \Rightarrow^+ \tau$.

- If we can produce the string τ from the string σ by applying n productions of G, with $n \geq 0$ (this includes the above and, in addition, the trivial case where $\sigma = \tau$), then we say that σ *produces* τ (or that τ *is derived from* σ), and express this $\sigma \Rightarrow^* \tau$.

- In terms of this notation, a **sentential form** is the goal or start symbol, or *any* string that can be derived from it, that is, any string σ such that $S \Rightarrow^* \sigma$.

- A grammar is called *recursive* if it permits derivations of the form $A \Rightarrow^+ \omega_1 A \omega_2$, (where $A \in N$, and $\omega_1, \omega_2 \in V^*$. More specifically, it is called *left recursive* if $A \Rightarrow^+ A \omega$ and *right recursive* if $A \Rightarrow^+ \omega A$.

- A grammar is *self-embedding* if it permits derivations of the form $A \Rightarrow^+ \omega_1 A \omega_2$, (where $A \in N$, and where $\omega_1, \omega_2 \in V^*$, but where ω_1 or ω_2 contain at least one terminal (that is $(\omega_1 \cap T) \cup (\omega_2 \cap T) \neq \emptyset$).

- Formally we can now define a language $L(G)$ produced by a grammar G by the relation

$$L(G) = \{ w \mid w \in T^*; S \Rightarrow^* w \}$$

5.5 Classic BNF notation for productions

As we have remarked, a production is a rule relating to a pair of strings, say γ and δ, specifying how one may be transformed into the other. This may be denoted $\gamma \rightarrow \delta$, and for simple theoretical grammars use is often made of this notation, using the conventions about the use of upper case letters for non-terminals and lower case ones for terminals. For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions for many years was to use an alternative notation invented by Backus, and first called Backus-Normal-Form. Later it was realized that it was not, strictly speaking, a "normal form", and was renamed Backus-Naur-Form. Backus and Naur were largely responsible for the *Algol 60 report* (Naur, 1960 and 1963), which was the first major attempt to specify the syntax of a programming language using this notation. Regardless of what the acronym really stands for, the notation is now universally known as **BNF**.

In classic BNF, a non-terminal is usually given a descriptive name, and is written in angle brackets to distinguish it from a terminal symbol. (Remember that non-terminals are used in the construction of sentences, although they do not actually appear in the final sentence.) In BNF, productions have the form

leftside → *definition*

Here "→" can be interpreted as "is defined as" or "produces" (in some texts the symbol ::= is used in preference to →). In such productions, both *leftside* and *definition* consist of a string concatenated from one or more terminals and non-terminals. In fact, in terms of our earlier notation

$$leftside \in (N \cup T)^{+}$$

and

$$definition \in (N \cup T)^{*}$$

although we must be more restrictive than that, for *leftside* must contain at least one non-terminal, so that we must also have

$$leftside \cap N \neq \emptyset$$

Frequently we find several productions with the same *leftside*, and these are often abbreviated by listing the *definitions* as a set of one or more alternatives, separated by a vertical bar symbol "|".

5.6 Simple examples

It will help to put the abstruse theory of the last two sections in better perspective if we consider two simple examples in some depth.

Our first example shows a grammar for a tiny subset of English itself. In full detail we have

```
G = {N , T , S , P}
N = {<sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }
T = { the , man , girl , boy , lecturer , he , she , drinks , sleeps ,
      mystifies , tall , thin , thirsty }
S = <sentence>
P = {  <sentence>       →   the <qualified noun> <verb>       (1)
                          | <pronoun> <verb>                  (2)
       <qualified noun> →  <adjective> <noun>                 (3)
       <noun>           →  man | girl | boy | lecturer        (4, 5, 6, 7)
       <pronoun>        →  he | she                           (8, 9)
       <verb>           →  talks | listens | mystifies        (10, 11, 12)
       <adjective>      →  tall | thin | sleepy               (13, 14, 15)
    }
```

The set of productions defines the non-terminal <sentence> as consisting of either the terminal "the" followed by a <qualified noun> followed by a <verb>, or as a <pronoun> followed by a <verb>. A <qualified noun> is an <adjective> followed by a <noun>, and a <noun> is one of the terminal symbols "man" or "girl" or "boy" or "lecturer". A <pronoun> is either of the terminals "he" or "she", while a <verb> is either "talks" or "listens" or "mystifies". Here <sentence>, <noun>, <qualified noun>, <pronoun>, <adjective> and <verb> are non-terminals. These do not appear in any sentence of the language, which includes such majestic prose as

the thin lecturer mystifies
he talks
the sleepy boy listens

From a grammar, one non-terminal is singled out as the so-called **goal** or **start symbol**. If we want to *generate* an arbitrary sentence we start with the goal symbol and successively replace each non-terminal on the right of the production defining that non-terminal, until all non-terminals have been removed. In the above example the symbol <sentence> is, as one would expect, the goal symbol.

Thus, for example, we could start with <sentence> and from this derive the sentential form

the <qualified noun> <verb>

In terms of the definitions of the last section we say that <sentence> *directly produces* "the <qualified noun> <verb>". If we now apply production 3 (<qualified noun> → <adjective> <noun>) we get the sentential form

the <adjective> <noun> <verb>

In terms of the definitions of the last section, "the <qualified noun> <verb>" directly produces "the <adjective> <noun> <verb>", while <sentence> has produced this sentential form in a non-trivial way. If we now follow this by applying production 14 (<adjective> → thin) we get the form

the thin <noun> <verb>

Application of production 10 (<verb> → talks) gets to the form

the thin <noun> talks

Finally, after applying production 6 (<noun> → boy) we get the sentence

the thin boy talks

The end result of all this is often represented by a tree, as in Figure 5.1, which shows a **phrase structure tree** or **parse tree** for our sentence. In this representation, the order in which the productions were used is not readily apparent, but it should now be clear why we speak of "terminals" and "non-terminals" in formal language theory - the leaves of such a tree are all terminals of the grammar; the interior nodes are all labelled by non-terminals.

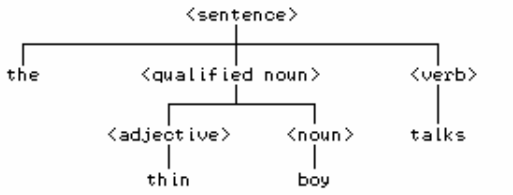


Figure 5.1 Parse tree for "the thin boy talks"

A moment's thought should reveal that there are many possible derivation paths from the goal or start symbol to the final sentence, depending on the order in which the productions are applied. It is convenient to be able to single out a particular derivation as being *the* derivation. This is generally called the **canonical derivation**, and although the choice is essentially arbitrary, the usual one is that where at each stage in the derivation the left-most non-terminal is the one that is replaced - this is called a **left canonical derivation**. (In a similar way we could define a **right canonical derivation**.)

Not only is it important to use grammars generatively in this way, it is also important - perhaps more so - to be able to take a given sentence and determine whether it is a valid member of the language - that is, to see whether it could have been obtained from the goal symbol by a suitable choice of derivations. When mere recognition is accompanied by the determination of the underlying tree structure, we speak of **parsing**. We shall have a lot more to say about this in later chapters; for the moment note that there are several ways in which we can attempt to solve the

problem. A fairly natural way is to start with the goal symbol and the sentence, and, by reading the sentence from left to right, to try to deduce which series of productions must have been applied.

Let us try this on the sentence

the thin boy talks

If we start with the goal <sentence> we can derive a wide variety of sentences. Some of these will arise if we choose to continue by using production 1, some if we choose production 2. By reading no further than "the" in the given sentence we can be fairly confident that we should try production 1.

<sentence> → the <qualified noun> <verb>.

In a sense we now have a residual input string "thin boy talks" which somehow must match <qualified noun> <verb>. We could now choose to substitute for <verb> or for <qualified noun>. Again limiting ourselves to working from left to right, our residual sentential form <qualified noun> <verb> must next be transformed into <adjective> <noun> <verb> by applying production 3.

In a sense we now have to match "thin boy talks" with a residual sentential form <adjective> <noun> <verb>. We could choose to substitute for any of <adjective>, <noun> or <verb>; if we read the input string from the left we see that by using production 14 we can reduce the problem of matching a residual input string "boy talks" to the residual sentential form <noun> <verb>. And so it goes; we need not labour a very simple point here.

The parsing problem is not always as easily solved as we have done. It is easy to see that the algorithms used to parse a sentence to see whether it can be derived from the goal symbol will be very different from algorithms that might be used to generate sentences (almost at random) starting from the start symbol. The methods used for successful parsing depend rather critically on the way in which the productions have been specified; for the moment we shall be content to examine a few sets of productions without worrying too much about how they were developed.

In BNF, a production may define a non-terminal recursively, so that the same non-terminal may occur on both the left and right sides of the → sign. For example, if the production for <qualified noun> were changed to

<qualified noun> → <noun> | <adjective> <qualified noun> (3a, 3b)

this would define a <qualified noun> as either a <noun>, or an <adjective> followed by a <qualified noun> (which in turn may be a <noun>, or an <adjective> followed by a <qualified noun> and so on). In the final analysis a <qualified noun> would give rise to zero or more <adjective>s followed by a <noun>. Of course, a recursive definition can only be useful provided that there is some way of terminating it. The single production

<qualified noun> → <adjective> <qualified noun> (3b)

is effectively quite useless on its own, and it is the alternative production

<qualified noun> → <noun> (3a)

which provides the means for terminating the recursion.

As a second example, consider a simple grammar for describing a somewhat restricted set of algebraic expressions:

```
G = {N , T , S , P}
N = {<goal> , <expression> , <term> , <factor> }
T = { a , b , c , - , * }
S = <goal>
P =
    <goal>        → <expression>                              (1)
    <expression>  → <term> | <expression> - <term>            (2, 3)
    <term>        → <factor> | <term> * <factor>              (4, 5)
    <factor>      → a | b | c                                 (6, 7, 8)
```

It is left as an easy exercise to show that it is possible to derive the string *a - b * c* using these productions, and that the corresponding phrase structure tree takes the form shown in Figure 5.2.

A point that we wish to stress here is that the construction of this tree has, happily, reflected the relative precedence of the multiplication and subtraction operations - assuming, of course, that the symbols * and - are to have implied meanings of "multiply" and "subtract" respectively. We should also point out that it is by no means obvious at this stage how one goes about designing a set of productions that not only describe the syntax of a programming language but also reflect some semantic meaning for the programs written in that language. Hopefully the reader can foresee that there will be a very decided advantage if such a choice *can* be made, and we shall have more to say about this in later sections.

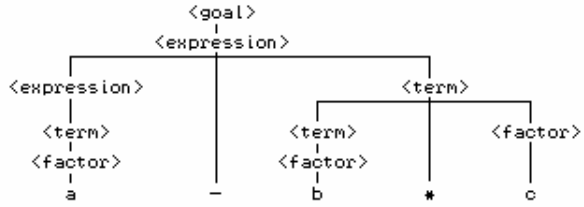


Figure 5.2 Parse tree for the expression a - b * c

Exercises

5.8 What would be the shortest sentence in the language defined by our first example? What would be the longest sentence? Would there be a difference if we used the alternative productions (3a, 3b)?

5.9 Draw the phrase structure trees that correspond to the expressions *a - b - c* and *a * b * c* using the second grammar.

5.10 Try to extend the grammar for expressions so as to incorporate the + and / operators.

5.7 Phrase structure and lexical structure

It should not take much to see that a set of productions for a real programming language grammar will usually divide into two distinct groups. In such languages we can distinguish between the productions that specify the **phrase structure** - the way in which the words or tokens of the

language are combined to form components of programs - and the productions that specify the **lexical structure** or **lexicon** - the way in which individual characters are combined to form such words or tokens. Some tokens are easily specified as simple constant strings standing for themselves. Others are more generic - lexical tokens such as identifiers, literal constants, and strings are themselves specified by means of productions (or, in many cases, by regular expressions).

As we have already hinted, the recognition of tokens for a real programming language is usually done by a scanner (lexical analyser) that returns these tokens to the parser (syntax analyser) on demand. The productions involving only individual characters on their right sides are thus the productions used by a sub-parser forming part of the lexical analyser, while the others are productions used by the main parser in the syntax analyser.

5.8 ε-productions

The alternatives for the right-hand side of a production usually consist of a string of one or more terminal and/or non-terminal symbols. At times it is useful to be able to derive an empty string, that is, one consisting of no symbols. This string is usually denoted by ε when it is necessary to reveal its presence explicitly. For example, the set of productions

```
<unsigned integer>   →   <digit> <rest of integer>
<rest of integer>    →   <digit> <rest of integer> | ε
<digit>              →   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

defines <rest of integer> as a sequence of zero or more <digit>s, and hence <unsigned integer> is defined as a sequence of one or more <digit>s. In terms of our earlier notation we should have

```
<rest of integer>  →  <digit>*
```

or

```
<unsigned integer>  →  <digit>+
```

The production

```
<rest of integer>  →  ε
```

is called a **null production**, or an ε-production, or sometimes a **lambda production** (from an alternative convention of using λ instead of ε for the null string). Applying a production of the form $L \rightarrow \varepsilon$ amounts to the erasure of the non-terminal L from a sentential form; for this reason such productions are sometimes called *erasures*. More generally, if for some string σ it is possible that

$$\sigma \Rightarrow^* \varepsilon$$

then we say that σ is *nullable*. A non-terminal L is said to be nullable if it has a production whose definition (right side) is nullable.

5.9 Extensions to BNF

Various simple extensions are often employed with BNF notation for the sake of increased readability and for the elimination of unnecessary recursion (which has a strange habit of confusing

people brought up on iteration). Recursion is often employed in BNF as a means of specifying simple repetition, as for example

```
<unsigned integer>  →  <digit> | <digit> <unsigned integer>
```

(which uses right recursion) or

```
<unsigned integer>  →  <digit> | <unsigned integer> <digit>
```

(which uses left recursion).

Then we often find several productions used to denote alternatives which are very similar, for example

```
<integer>           →  <unsigned integer> | <sign> <unsigned integer>
<unsigned integer>  →  <digit> | <digit> <unsigned integer>
<sign>              →  + | -
```

using six productions (besides the omitted obvious ones for <digit>) to specify the form of an <integer>.

The extensions introduced to simplify these constructions lead to what is known as **EBNF** (Extended BNF). There have been many variations on this, most of them inspired by the metasymbols used for regular expressions. Thus we might find the use of the Kleene closure operators to denote repetition of a symbol zero or more times, and the use of round brackets or parentheses ( ) to group items together.

Using these ideas we might define an integer by

```
<integer>           →  <sign> <unsigned integer>
<unsigned integer>  →  <digit> ( <digit> )*
<sign>              →  + | - | ε
```

or even by

```
<integer>    →   ( + | - | ε )  <digit> ( <digit> )*
```

which is, of course, nothing other than a regular expression anyway. In fact, a language that can be expressed as a regular expression can always be expressed in a single EBNF expression.

5.9.1 Wirth's EBNF notation

In defining Pascal and Modula-2, Wirth came up with one of these many variations on BNF which has now become rather widely used (Wirth, 1977). Further metasymbols are used, so as to express more succinctly the many situations that otherwise require combinations of the Kleene closure operators and the ε string. In addition, further simplifications are introduced to facilitate the automatic processing of productions by parser generators such as we shall discuss in a later section. In this notation for EBNF:

`Non-terminals`	are written as single words, as in *VarDeclaration* (rather than the <Var Declaration> of our previous notation)
`Terminals`	are all written in quotes, as in "`BEGIN`" (rather than as themselves, as in BNF)
`|`	is used, as before, to denote alternatives
`(  )`	(parentheses) are used to denote grouping
`[  ]`	(brackets) are used to denote the optional appearance of a

	symbol or group of symbols
`{ }`	(braces) are used to denote optional repetition of a symbol or group of symbols
`=`	is used in place of the ::= or → symbol
`.`	is used to denote the end of each production
`(* *)`	are used in some extensions to allow comments
ε	can be handled by using the [ ] notation
`spaces`	are essentially insignificant.

For example

```
Integer         =  Sign UnsignedInteger .
UnsignedInteger =  digit  {  digit  } .
Sign            =  [ "+" | "-" ] .
digit           =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

The effect is that non-terminals are less "noisy" than in the earlier forms of BNF, while terminals are "noisier". Many grammars used to define programming language employ far more non-terminals than terminals, so this is often advantageous. Furthermore, since the terminals and non-terminals are textually easily distinguishable, it is usually adequate to give only the set of productions *P* when writing down a grammar, and not the complete quadruple { *N, T, S, P* }.

As another example of the use of this notation we show how to describe a set of EBNF productions in EBNF itself:

```
EBNF        =  { Production } .
Production  =  nonterminal "=" Expression "." .
Expression  =  Term { "|" Term } .
Term        =  Factor { Factor } .
Factor      =      nonterminal | terminal | "[" Expression "]"
                | "(" Expression ")" | "{" Expression "}" .
nonterminal =  letter { letter } .
terminal    =  "'" character { character } "'" | '"' character { character } '"' .
character   =  (* implementation defined *)  .
```

Here we have chosen to spell *nonterminal* and *terminal* in lower case throughout to emphasize that they are lexical non-terminals of a slightly different status from the others like *Production, Expression, Term* and *Factor*.

A variation on the use of braces allows the (otherwise impossible) specification of a limit on the number of times a symbol may be repeated - for example to express that an identifier in Fortran may have a maximum of six characters. This is done by writing the lower and upper limits as sub- and super-scripts to the right of the curly braces, as for example

$$FortranIdentifier \rightarrow letter \{ letter \mid digit \}_{0}^{5}$$

5.9.2 Semantic overtones

Sometimes productions are developed to give semantic overtones. As we shall see in a later section, this leads more easily towards the possibility of extending or *attributing* the grammar to incorporate a formal semantic specification along with the syntactic specification. For example, in describing Modula-2, where expressions and identifiers fall into various classes at the static semantic level, we might find among a large set of productions:

```
ConstDeclarations    =  "CONST"
                           ConstIdentifier "=" ConstExpression ";"
                           { ConstIdentifier "=" ConstExpression ";" } .
ConstIdentifier      =  identifier .
```

```
ConstExpression      =  Expression .
```

5.9.3 The British Standard for EBNF

The British Standards Institute has a published standard for EBNF (BS6154 of 1981). The BSI standard notation is noisier than Wirth's one: elements of the productions are separated by commas, productions are terminated by semicolons, and spaces become insignificant. This means that compound words like *ConstIdentifier* are unnecessary, and can be written as separate words. An example in BSI notation follows:

```
Constant Declarations  =  "CONST",
                               Constant Identifier, "=", Constant Expression, ";",
                             { Constant Identifier, "=", Constant Expression, ";" }  ;
Constant Identifier    =  identifier ;
Constant Expression    =  Expression ;
```

5.9.4 Lexical and phrase structure emphasis

We have already commented that real programming language grammars have a need to specify phrase structure as well as lexical structure. Sometimes the distinction between "lexical" and "syntactic" elements is taken to great lengths. For example we might find:

```
ConstDeclarations    =  constSym
                           ConstIdentifier equals ConstExpression semicolon
                           { ConstIdentifier equals ConstExpression semicolon } .
```

with productions like

```
  constSym             =  "CONST" .
  semicolon            =  ";" .
  equals               =  "=" .
```

and so on. This may seem rather long-winded, but there are occasional advantages, for example in allowing alternatives for limited character set machines, as in

```
  leftBracket          =  "["  |  "(." .
  pointerSym           =  "^"  |  "@" .
```

as is used in some Pascal systems.

5.9.5 Cocol

The reader will recall from Chapter 2 that compiler writers often make use of compiler generators to assist with the automated construction of parts of a compiler. Such tools usually take as input an augmented description of a grammar, one usually based on a variant of the EBNF notations we have just been discussing. We stress that far more is required to construct a compiler than a description of syntax - which is, essentially, all that EBNF can provide. In later chapters we shall describe the use of a specific compiler generator, Coco/R, a product that originated at the University of Linz in Austria (Rechenberg and Mössenböck, 1989, Mössenböck, 1990a,b). The name Coco/R is derived from "**Co**mpiler-**Co**mpiler/**R**ecursive descent. A variant of Wirth's EBNF known as Cocol/R is used to define the input to Coco/R, and is the notation we shall prefer in the rest of this text (to avoid confusion between two very similar acronyms we shall simply refer to Cocol/R as Cocol). Cocol draws a clear distinction between lexical and phrase structure, and also makes clear provision for describing the character sets from which lexical tokens are constructed.

A simple example will show the main features of a Cocol description. The example describes a calculator that is intended to process a sequence of simple four-function calculations involving decimal or hexadecimal whole numbers, for example `3 + 4 * 8 =` or `$3F / 7 + $1AF =`.

```
COMPILER Calculator

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = digit + "ABCDEF" .

IGNORE CHR(1) .. CHR(31)

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

PRODUCTIONS
  Calculator = { Expression "=" } .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = decNumber | hexNumber .
END Calculator.
```

The `CHARACTERS` section describes the set of characters that can appear in decimal or hexadecimal digit strings - the right sides of these productions are to be interpreted as defining sets. The `TOKENS` section describes the valid forms that decimal and hexadecimal numbers may take - but notice that we do not, at this stage, indicate how the values of these numbers are to be computed from the digits. The `PRODUCTIONS` section describes the phrase structure of the calculations themselves - again without indicating how the results of the calculations are to be obtained.

At this stage it will probably come as no surprise to the reader to learn that Cocol, the language of the input to Coco/R, can itself be described by a grammar - and, indeed, we may write this grammar in a way that it could be processed by Coco/R itself. (Using Coco/R to process its own grammar is, of course, just another example of the bootstrapping techniques discussed in Chapter 3; Coco/R is another good example of a self-compiling compiler). A full description of Coco/R and Cocol appears later in this text, and while the finer points of this may currently be beyond the reader's comprehension, the following simplified description will suffice to show the syntactic elements of most importance:

```
COMPILER Cocol

CHARACTERS
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  tab          = CHR(9) .
  cr           = CHR(13) .
  lf           = CHR(10) .
  noQuote2     = ANY - '"' - cr - lf .
  noQuote1     = ANY - "'" - cr - lf .

IGNORE tab + cr + lf

TOKENS
  identifier   = letter { letter | digit } .
  string       = '"' { noQuote2 } '"' | "'" { noQuote1 } "'" .
  number       = digit { digit } .

PRODUCTIONS
  Cocol        = "COMPILER"  Goal
                    [ Characters ]
                    [ Ignorable ]
                    [ Tokens ]
                    Productions
                 "END" Goal "." .
  Goal         = identifier .

  Characters   = "CHARACTERS" { NamedCharSet } .
  NamedCharSet = SetIdent "=" CharacterSet "." .
  CharacterSet = SimpleSet { "+" SimpleSet | "-" SimpleSet } .
  SimpleSet    = SetIdent | string | SingleChar [ ".." SingleChar ] | "ANY" .
  SingleChar   = "CHR" "(" number ")" .
  SetIdent     = identifier .

  Ignorable    = "IGNORE" CharacterSet .

  Tokens       = "TOKENS" { Token } .
  Token        = TokenIdent "=" TokenExpr "."  .
```

```
  TokenExpr    = TokenTerm { "|" TokenTerm } .
  TokenTerm    = TokenFactor { TokenFactor } [ "CONTEXT" "(" TokenExpr ")" ] .
  TokenFactor  =   TokenSymbol | "(" TokenExpr ")" | "[" TokenExpr "]"
                 | "{" TokenExpr "}" .
  TokenSymbol  = SetIdent | string .
  TokenIdent   = identifier .

  Productions  = "PRODUCTIONS" { Production } .
  Production   = NonTerminal "=" Expression  "." .
  Expression   = Term { "|" Term } .
  Term         = Factor { Factor } .
  Factor       =   Symbol | "(" Expression ")" | "[" Expression "]"
                 | "{" Expression "}" .
  Symbol       = string | NonTerminal | TokenIdent .
  NonTerminal  = identifier .

END Cocol.
```

The following points are worth emphasizing:

- The productions in the `TOKENS` section specify identifiers, strings and numbers in the usual simple way.

- The first production (for Cocol) shows the overall form of a grammar description as consisting of four sections, the first three of which are all optional (although they are usually present in practice).

- The productions for `CharacterSets` show how character sets may be given names (*SetIdents*) and values (of *SimpleSets*).

- The production for `Ignorable` allows certain characters - typically line feeds and other unimportant characters - to be included in a set that will simply be ignored by the scanner when it searches for and recognizes tokens.

- The productions for `Tokens` show how tokens (terminal classes) may be named (*TokenIdents*) and defined by expressions in EBNF. Careful study of the semantic overtones of these productions will show that they are not self-embedding - that is, one token may not be defined in terms of another token, but only as a quoted string, or in terms of characters chosen from the named character sets defined in the `CHARACTERS` section. This amounts, in effect, to defining these tokens by means of regular expressions, even though the notation used is not the same as that given for regular expressions in section 5.3.

- The productions for `Productions` show how we define the phrase structure by naming *NonTerminals* and expressing their productions in EBNF. Notice that here we *are* allowed to have self-embedding and recursive productions. Although terminals may again be specified directly as strings, we are not allowed to use the names of character sets as symbols in the productions.

- Although it is not specified by the grammar above, one non-terminal must have the same identifier name as the grammar itself to act as the goal symbol (and, of course, all identifiers must be "declared" properly).

- It is possible to write input in Cocol that is syntactically correct (in terms of the grammar above) but which cannot be fully processed by Coco/R because it does not satisfy other constraints. This topic will be discussed further in later sections.

We stress again that Coco/R input really specifies *two* grammars. One is the grammar specifying the non- terminals for the lexical analyser (`TOKENS`) and the other specifies non-terminals for the

higher level phrase structure grammar used by the syntax analyser (PRODUCTIONS). However, terminals may also be implicitly declared in the productions section. So the following, in one sense, may appear to be equivalent:

```
COMPILER Sample   (* one *)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  ident = letter { letter } .

PRODUCTIONS
  Sample = "BEGIN" ident ":=" ident "END" .

END Sample .
```

```
COMPILER Sample   (* two *)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  Letter = letter .

PRODUCTIONS
  Sample = "BEGIN" Ident ":=" Ident "END" .
  Ident  = Letter { Letter } .

END Sample .
```

```
COMPILER Sample   (* three *)

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  ident = letter { letter } .
  begin = "BEGIN" .
  end   = "END" .
  becomes = ":=" .

PRODUCTIONS
  Sample = begin ident becomes ident end .

END Sample .
```

Actually they are not quite the same. Since Coco/R always ignores spaces (other than in strings), the second one would treat the input

```
A C E := S P A D E
```

as the first would treat the input

```
ACE := SPADE
```

The best simple rule seems to be that one should declare under TOKENS any class of symbol that has to be recognized as a contiguous string of characters, and of which there may be several instances (this includes entities like identifiers, numbers, and strings) - as well as special character terminals (like EOL) that cannot be graphically represented as quoted characters. Reserved keywords and symbols like ":=" are probably best introduced as terminals implicitly declared in the PRODUCTIONS section. Thus grammar (1) above is probably the best so far as Coco/R is concerned.

Exercises

5.11 Develop simple grammars to describe each of the following

(a) A person's name, with optional title and qualifications (if any), for example

S.B. Terry , BSc
Master Kenneth David Terry
Helen Margaret Alice Terry

(b) A railway goods train, with one (or more) locomotives, several varieties of trucks, and a guard's van at the rear.

(c) A mixed passenger and goods train, with one (or more) locomotives, then one or more goods trucks, followed either by a guard's van, or by one or more passenger coaches, the last of which should be a passenger brake van. In the interests of safety, try to build in a regulation to the effect that fuel trucks may not be marshalled immediately behind the locomotive, or immediately in front of a passenger coach.

(d) A book, with covers, contents, chapters and an index.

(e) A shopping list, with one or more items, for example

3 Practical assignments
124 bottles Castle Lager
12 cases Rhine Wine
large box aspirins

(f) Input to a postfix (reverse Polish) calculator. In postfix notation, brackets are not used, but instead the operators are placed after the operands.

For example,

```
infix expression            reverse Polish equivalent

6 + 9 =                     6 9 + =
(a + b) * (c + d)           a b + c d + *
```

(g) A message in Morse code.

(h) Unix or MS-DOS file specifiers.

(i) Numbers expressed in Roman numerals.

(j) Boolean expressions incorporating conjunction (OR), disjunction (AND) and negation (NOT).

5.12 Develop a Cocol grammar using only BNF-style productions that defines the rules for expressing a set of BNF productions.

5.13 Develop a Cocol grammar using only BNF-style productions that defines the rules for expressing a set of EBNF productions.

5.14 Develop an EBNF grammar that defines regular expressions as described in section 5.3.

5.15 What real practical advantage does the Wirth notation using [ ] and { } afford over the use of the Kleene closure symbols?

5.16 In yet another variation on EBNF ε can be written into an empty right side of a production explicitly, in addition to being handled by using the [ ] notation, for example:

```
Sign = "+" | "-" | .  (* the ε or null is between the last | and . *)
```

Productions like this cannot be described by the productions for EBNF given in section 5.9.1. Develop a Cocol grammar that describes EBNF productions that *do* allow an empty string to appear implicitly.

5.17 The local Senior Citizens Association make a feature of Friday evenings, when they employ a mediocre group to play for dancing. At such functions the band perform a number of selections, interspersed with periods of silence which are put to other good use. The band have only four kinds of selection at present. The first of these consists of waltzes - such a selection always starts with a slow waltz, which may be followed by several more slow waltzes, and finally (but only if the mood of the evening demands it) by one or more fast waltzes. The second type of selection consists of several Rock'n'Roll numbers. The third is a medley, consisting of a number of tunes of any sort played in any order. The last is the infamous "Paul Jones", which is a special medley in which every second tune is "Here we go round the mulberry bush". During the playing of this, the dancers all pretend to change partners, in some cases actually succeeding in doing so. Develop a grammar which describes the form that the evening assumes.

5.18 Scottish pipe bands often compete at events called Highland Gatherings where three forms of competition are traditionally mounted. There is the so-called "Slow into Quick March" competition, in which each band plays a single Slow March followed by a single Quick March. There is the so-called "March, Strathspey and Reel" competition, where each band plays a single Quick March, followed by a single Strathspey, and then by a single Reel; this set may optionally be followed by a further Quick March. And there is also the "Medley", in which a band plays a selection of tunes in almost any order. Each tune fall into one of the categories of March, Strathspey, Reel, Slow March, Jig and Hornpipe but, by tradition, a group of one or more Strathspeys within such a medley is always followed by a group of one or more Reels.

Develop a grammar to describe the activity at a Highland Gathering at which a number of competitions are held, and in each of which at least one band performs. Competitions are held in one category at a time. Regard concepts like "March", "Reel" and so on as terminals - in fact there are many different possible tunes of each sort, but you may have to be a piper to recognize one tune from another.

5.19 Here is an extract from the index of my forthcoming bestseller "Hacking out a Degree":

abstract class 12, 45
abstraction, data 165
advantages of Modula-2 1-99, 100-500, Appendix 4
aegrotat examinations -- see unethical doctors
class attendance, intolerable 745
deadlines, compiler course -- see sunrise
horrible design (C and C++) 34, 45, 85-96
lectures, missed 1, 3, 5-9, 12, 14-17, 21-25, 28
recursion -- see recursion
senility, onset of 21-24, 105

subminimum 30
supplementary exams 45 - 49
wasted years 1996-1998

Develop a grammar that describes this form of index.

5.20 You may be familiar with the "make" facility that is found on Unix (and sometimes on MS-DOS) for program development. A "make file" consists of input to the `make` command that typically allows a system to be re-built correctly after possibly modifying some of its component parts. A typical example for a system involving C++ compilation is shown below. Develop a grammar that describes the sort of make files that you may have used in your own program development.

```
# makefile for maintaining my compiler
CFLAGS = -Wall
CC     = g++
HDRS   = parser.h scanner.h generator.h
SRCS   = compiler.cpp \
         parser.cpp scanner.cpp generator.cpp
OBJS   = compiler.o parser.o scanner.o generator.o

%.o: %.cpp $(HDRS)
        $(CC) -c $(CFLAGS) $<

all:    compiler

new:    clean compiler

cln:    $(OBJS)
        $(CC) -o cln $(CFLAGS) $(OBJS)

clean:
        rm *.o
        rm compiler
```

5.21 C programmers should be familiar with the use of the standard functions `scanf` and `printf` for performing input and output. Typical calls to these functions are

```
scanf("%d %s %c", &n, string, &ch);
printf("Total = %-10.4d\nProfit = %d\%%\n", total, profit);
```

in which the first argument is usually a literal string incorporating various specialized format specifiers describing how the remaining arguments are to be processed.

Develop a grammar that describes such statements as fully as you can. For simplicity restrict yourself to the situation where any arguments after the first refer to simple variables.

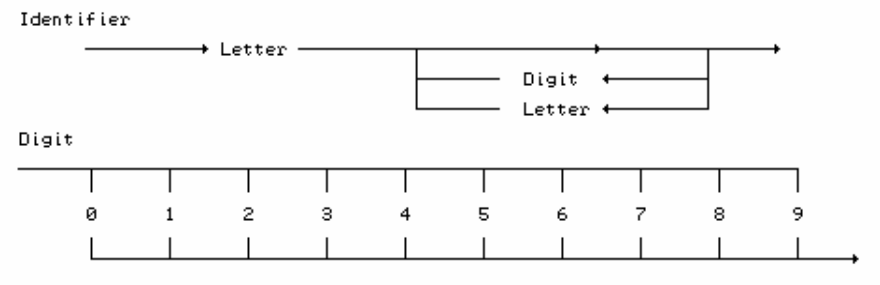
Further reading

The use of BNF and EBNF notation is covered thoroughly in all good books on compilers and syntax analysis. Particularly useful insight will be found in the books by Watt (1991), Pittman and Peters (1992) and Gough (1988).

5.10 Syntax diagrams

An entirely different method of syntax definition is by means of the graphic representation known

as syntax diagrams, syntax charts, or sometimes "railroad diagrams". These have been used to define the syntax of Pascal, Modula-2 and Fortran 77. The rules take the form of flow diagrams, the possible paths representing the possible sequences of symbols. One starts at the left of a diagram, and traces a path which may incorporate terminals, or incorporate transfers to other diagrams if a word is reached that corresponds to a non-terminal. For example, an identifier might be defined by



with a similar diagram applying to `Letter`, which we can safely assume readers to be intelligent enough to draw for themselves.

Exercises

5.22 Attempt to express some of the solutions to previous exercises in terms of syntax diagrams.

5.11 Formal treatment of semantics

As yet we have made no serious attempt to describe the semantics of programs written in any of our "languages", and have just assumed that these would be self-evident to a reader who already has come to terms with at least one imperative language. In one sense this is satisfactory for our purposes, but in principle it is highly unsatisfactory not to have a simple, yet rigidly formal means of specifying the semantics of a language. In this section we wish to touch very briefly on ways in which this might be achieved.

We have already commented that the division between syntax and semantics is not always clear-cut, something which may be exacerbated by the tendency to specify productions using names with clearly semantic overtones, and whose sentential forms already reflect meanings to be attached to operator precedence and so on. When specifying semantics a distinction is often attempted between what is termed **static semantics** - features which, in effect, mean something that can be checked at compile-time, such as the requirement that one may not branch into the middle of a **procedure**, or that assignment may only be attempted if type checking has been satisfied - and **dynamic semantics** - features that really only have meaning at run-time, such as the effect of a branch statement on the flow of control, or the effect of an assignment statement on elements of storage.

Historically, attempts formally to specify semantics did not meet with the same early success as those which culminated in the development of BNF notation for specifying syntax, and we find that the semantics of many, if not most, common programming languages have been explained in terms of a natural language document, often regrettably imprecise, invariably loaded with jargon, and difficult to follow (even when one has learned the jargon). It will suffice to give two examples:

(a) In a draft description of Pascal, the syntax of the **with** statement was defined by

```
      <with-statement> ::=  with <record-variable-list> do <statement>
<record-variable-list> ::=  <record-variable> { , <record-variable> }
<variable-identifier>  ::=  <field-identifier>
```

with the commentary that

"The occurrence of a <record-variable> in the <record-variable-list> is a defining occurrence of its <field-identifier>s as <variable-identifier>s for the <with-statement> in which the <record-variable-list> occurs."

The reader might be forgiven for finding this awkward, especially in the way it indicates that within the <statement> the <field-identifier>s may be used as though they were <variable-identifier>s.

(b) In the same description we find the **while** statement described by

<while-statement> : : = **while** <Boolean-expression> **do** <statement>

with the commentary that

"The <statement> is repeatedly executed while the <Boolean-expression> yields the value TRUE. If its value is FALSE at the beginning, the <statement> is not executed at all. The <while- statement>

while *b* **do** *body*

is equivalent to

if *b* **then repeat** *body* **until not** *b*."

If one is to be very critical, one might be forgiven for wondering what exactly is meant by "beginning" (does it mean the beginning of the program, or of execution of this one part of the program). One might also conclude, especially from all the emphasis given to the effect when the <Boolean-expression> is initially FALSE, that in that case the <while-statement> is completely equivalent to an empty statement. This is not necessarily true, for evaluation of the <Boolean-expression> might require calls to a function which has side-effects; nowhere (at least in the vicinity of this description) was this point mentioned.

The net effect of such imprecision and obfuscation is that users of a language often resort to writing simple test programs to help them understand language features, that is to say, they use the operation of the machine itself to explain the language. This is a technique which can be disastrous on at least two scores. In the first place, the test examples may be incomplete, or too special, and only a half-truth will be gleaned. Secondly, and perhaps more fundamentally, one is then confusing an abstract language with one concrete implementation of that language. Since implementations may be error prone, incomplete, or, as often happens, may have extensions that do not form part of the standardized language at all, the possibilities for misconception are enormous.

However, one approach to formal specification, known as **operational semantics** essentially refines this ad-hoc arrangement. To avoid the problems mentioned above, the (written) specification usually describes the action of a program construction in terms of the changes in state of an abstract machine which is supposed to be executing the construction. This method was used to specify the language PL/I, using the metalanguage **VDL** (Vienna Definition Language). Of course,

to understand such specifications, the reader has to understand the definition of the abstract machine, and not only might this be confusingly theoretical, it might also be quite unlike the actual machines which he or she has encountered. As in all semantic descriptions, one is simply shifting the problem of "meaning" from one area to another. Another drawback of this approach is that it tends to obscure the semantics with a great detail of what is essentially useful knowledge for the implementor of the language, but almost irrelevant for the user of the same.

Another approach makes use of **attribute grammars**, in which the syntactic description (in terms of EBNF) is augmented by a set of distinct attributes *V* (each one associated with a single terminal or non-terminal) and a set of assertions or predicates involving these attributes, each assertion being associated with a single production. We shall return to this approach in a later chapter, for it forms the basis of practical applications of several compiler generators, among them Coco/R.

Other approaches taken to specifying semantics tend to rely rather more heavily on mathematical logic and mathematical notation, and for this reason may be almost impossible to understand if the programmer is one of the many thousands whose mathematical background is comparatively weak. **Denotational semantics**, for example defines programs in terms of mappings into mathematical operations and constructs: a program is simply a function that maps its input data to its output data, and its individual component statements are functions that map an environment and store to an updated store. A variant of this, **VDM** (Vienna Definition Method), has been used in formal specifications of Ada, Algol-60, Pascal and Modula-2. These specifications are long and difficult to follow (that for Modula-2 runs to some 700 pages).

Another mathematically based method, which was used by Hoare and Wirth (1973) to specify the semantics of most of Pascal, uses so-called **axiomatic semantics**, and it is worth a slight digression to examine the notation used. It is particularly apposite when taken in conjunction with the subject of **program proving**, but, as will become apparent, rather limited in the way in which it specifies what a program actually seems to be doing.

In the notation, *S* is used to represent a statement or statement sequence, and letters like *P, Q* and *R* are used to represent **predicates**, that is, the logical values of Boolean variables or expressions. A notation like

$$\{ P \} S \{ Q \}$$

denotes a so-called **inductive expression**, and is intended to convey that if *P* is true before *S* is executed, then *Q* will be true after *S* terminates (assuming that it does terminate, which may not always happen).

P is often called the **precondition** and *Q* the **postcondition** of *S*. Such inductive expressions may be concatenated with logical operations like $\wedge$ (*and*) and $\neg$ (*not*) and $\Rightarrow$ (*implies*) to give expressions like

$$\{ P \} S_1 \{ Q \} \wedge \{ Q \} S_2 \{ R \}$$

from which one can infer that

$$\{ P \} S_1 ; S_2 \{ R \}$$

which is written more succinctly as a **rule of inference**

$$\frac{\{ P \}\ S_1\ \{ Q \} \wedge \{ Q \}\ S_2\ \{ R \}}{\{ P \}\ S_1 ; S_2\ \{ R \}}$$

Expressions like

$P \Rightarrow Q$ and $\{ Q \}\ S\ \{ R \}$

and

$\{ P \}\ S\ \{ Q \}$ and $Q \Rightarrow R$

lead to the **consequence rules**

$$\frac{P \Rightarrow Q \text{ and } \{ Q \}\ S\ \{ R \}}{\{ P \}\ S\ \{ R \}}$$

and

$$\frac{\{ P \}\ S\ \{ Q \} \text{ and } Q \Rightarrow R}{\{ P \}\ S\ \{ R \}}$$

In these rules, the top line is called the **antecedent** and the bottom one is called the **consequent**; so far as program proving is concerned, to prove the truth of the consequent it is necessary only to prove the truth of the antecedent.

In terms of this notation one can write down rules for nearly all of Pascal remarkably tersely. For example, the **while** statement can be described by

$$\frac{\{ P \wedge \mathrm{B} \}\ S\ \{ P \}}{\{ P \}\ \textbf{while}\ \mathrm{B}\ \textbf{do}\ S\ \{ P \wedge \neg\, \mathrm{B} \}}$$

and the **if** statements by

$$\frac{\{ P \wedge \mathrm{B} \}\ S\ \{ Q \} \text{ and } P \wedge \neg\, \mathrm{B} \Rightarrow Q}{\{ P \}\ \textbf{if}\ \mathrm{B}\ \textbf{then}\ S\ \{ Q \}}$$

$$\frac{\{ P \wedge \mathrm{B} \}\ S_1\ \{ Q \} \text{ and } \{ P \wedge \neg\, \mathrm{B} \}\ S_2\ \{ Q \}}{\{ P \}\ \textbf{if}\ \mathrm{B}\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{ Q \}}$$

With a little reflection one can understand this notation quite easily, but it has its drawbacks. Firstly, the rules given are valid only if the evaluation of *B* proceeds without side-effects (compare the discussion earlier). Secondly, there seems to be no explicit description of what the machine implementing the program actually does to alter its state - the idea of "repetition" in the rule for the **while** statement probably does not exactly strike the reader as obvious.

Further reading

In what follows we shall, perhaps cowardly, rely heavily on the reader’s intuitive grasp of semantics. However, the keen reader might like to follow up the ideas germinated here. So far as natural language descriptions go, a draft description of the Pascal Standard is to be found in the article by Addyman *et al* (1979). This was later modified to become the ISO Pascal Standard, known variously as ISO 7185 and BS 6192, published by the British Standards Institute, London (a copy is given as an appendix to the book by Wilson and Addyman (1982)). A most readable guide to the Pascal Standard was later produced by Cooper (1983). Until a standard for C++ is completed, the most precise description of C++ is probably the "ARM" (Annotated Reference Manual) by Ellis and Stroustrup (1990), but C++ has not yet stabilized fully (in fact the standard appeared shortly after this book was published). In his book, Brinch Hansen (1983) has a very interesting chapter on the problems he encountered in trying to specify Edison completely and concisely.

The reader interested in the more mathematically based approach will find useful introductions in the very readable books by McGettrick (1980) and Watt (1991). Descriptions of VDM and specifications of languages using it are to be found in the book by Bjorner and Jones (1982). Finally, the text by Pittman and Peters (1992) makes extensive use of attribute grammars.

6 SIMPLE ASSEMBLERS

In this chapter we shall be concerned with the implementation of simple assembler language translator programs. We assume that the reader already has some experience in programming at the assembler level; readers who do not will find excellent discussions of this topic in the books by Wakerly (1981) and MacCabe (1993). To distinguish between programs written in "assembler code", and the "assembler program" which translates these, we shall use the convention that ASSEMBLER means the language and "assembler" means the translator.

The basic purpose of an assembler is to translate ASSEMBLER language mnemonics into binary or hexadecimal machine code. Some assemblers do little more than this, but most modern assemblers offer a variety of additional features, and the boundary between assemblers and compilers has become somewhat blurred.

6.1 A simple ASSEMBLER language

Rather than use an assembler for a real machine, we shall implement one for a rudimentary ASSEMBLER language for the hypothetical single-accumulator machine discussed in section 4.3.

An example of a program in our proposed language is given below, along with its equivalent object code. We have, as is conventional, used hexadecimal notation for the object code; numeric values in the source have been specified in decimal.

```
Assembler 1.0 on 01/06/96 at 17:40:45

00                  BEG                 ; count the bits in a number
00    0A            INI                 ; Read(A)
01          LOOP                        ; REPEAT
01    16            SHR                 ;  A := A DIV 2
02    3A 0D         BCC     EVEN        ;  IF A MOD 2 # 0 THEN
04    1E 13         STA     TEMP        ;    TEMP := A
06    19 14         LDA     BITS
08    05            INC
09    1E 14         STA     BITS        ;    BITS := BITS + 1
0B    19 13         LDA     TEMP        ;    A := TEMP
0D    37 01  EVEN   BNZ     LOOP        ; UNTIL A = 0
0F    19 14         LDA     BITS        ;
11    0E            OTI                 ; Write(BITS)
12    18            HLT                 ; terminate execution
13          TEMP    DS      1           ; VAR TEMP : BYTE
14    00    BITS    DC      0           ;     BITS : BYTE
15                  END
```

ASSEMBLER programs like this usually consist of a sequence of statements or instructions, written one to a line. These statements fall into two main classes.

Firstly, there are the **executable instructions** that correspond directly to executable code. These can be recognized immediately by the presence of a distinctive **mnemonic** for an **opcode**. For our machine these executable instructions divide further into two classes: there are those that require an **address** or operand as part of the instruction (as in `STA TEMP`) and occupy two bytes of object code, and there are those that stand alone (like `INI` and `HLT`). When it is necessary to refer to such statements elsewhere, they may be labelled with an introductory distinctive **label** identifier of the programmer's choice (as in `EVEN BNZ LOOP`), and may include a **comment**, extending from an introductory semicolon to the end of a line.

The address or operand for those instructions that requires them is denoted most simply by either a numeric literal, or by an identifier of the programmer's choice. Such identifiers usually correspond to the ones that are used to label statements - when an identifier is used to label a statement itself we speak of a **defining occurrence** of a label; when an identifier appears as an address or operand we speak of an **applied occurrence** of a label.

The second class of statement includes the **directives**. In source form these appear to be deceptively similar to executable instructions - they are often introduced by a label, terminated with a comment, and have what may appear to be mnemonic and address components. However, directives have a rather different role to play. They do not generally correspond to operations that will form part of the code that is to be executed at *run-time*, but rather denote actions that direct the action of the assembler at *compile-time* - for example, indicating where in memory a block of code or data is to be located when the object code is later loaded, or indicating that a block of memory is to be preset with literal values, or that a name is to be given to a literal to enhance readability.

For our ASSEMBLER we shall introduce the following directives and their associated compile-time semantics, as a representative sample of those found in more sophisticated languages:

```
 Label Mnemonic   Address    Effect

not used   BEG    not used   Mark the beginning of the code
not used   END    not used   Mark the end of the code
not used   ORG    location   Specify location where the following code
                             is to be loaded
optional   DC     value      Define an (optionally labelled) byte,
                             to have a specified initial value
optional   DS     length     Reserve length bytes (optional label associated
                             with the first byte)
name       EQU    value      Set name to be a synonym for the given value
```

Besides lines that contain a full statement, most assemblers usually permit incomplete lines. These may be completely blank (so as to enhance readability), or may contain only a label, or may contain only a comment, or may contain only a label and a comment.

Our first task might usefully be to try to find a grammar that describes this (and similar) programs. This can be done in several ways. Our informal description has already highlighted various syntactic classes that will be useful in specifying the phrase structure of our programs, as well as various token classes that a scanner may need to recognize as part of the assembly process. One possible grammar - which leaves the phrase structure very loosely defined - is given below. This has been expressed in Cocol, the EBNF variant introduced in section 5.9.5.

```
COMPILER ASM

  CHARACTERS
    eol        = CHR(13) .
    letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
    digit      = "0123456789" .
    printable  = CHR(32) .. CHR(127) .

  IGNORE  CHR(9) .. CHR(12)

  TOKENS
    number     = digit { digit } .
    identifier = letter { letter | digit } .
    EOL        = eol .
    comment    = ";" { printable } .

  PRODUCTIONS
    ASM        = { Statement } EOF .
    Statement  = [ Label ] [ Mnemonic [ Address ] ] [ comment ] EOL .
    Address    = Label | number .
    Mnemonic   = identifier .
    Label      = identifier .
END ASM.
```

This grammar has the advantage of simplicity, but makes no proper distinction between directives and executable statements, nor does it indicate which statements *require* labels or address fields. It is possible to draw these distinctions quite easily if we introduce a few more non-terminals into the phrase structure grammar:

```
COMPILER ASM

  CHARACTERS
    eol        = CHR(13) .
    letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
    digit      = "0123456789" .
    printable  = CHR(32) .. CHR(127) .

  IGNORE  CHR(9) .. CHR(12)

  TOKENS
    number     = digit { digit } .
    identifier = letter { letter | digit } .
    EOL        = eol .
    comment    = ";" { printable } .

  PRODUCTIONS
    ASM               = StatementSequence "END" EOF .
    StatementSequence = { Statement [ comment ] EOL } .
    Statement         = Executable | Directive .
    Executable        = [ Label ] [ OneByteOp | TwoByteOp Address ] .
    OneByteOp         = "HLT" | "PSH" | "POP"  (* | . . . . etc *) .
    TwoByteOp         = "LDA" | "LDX" | "LDI"  (* | . . . . etc *) .
    Address           = Label | number .
    Directive         =   Label "EQU" KnownAddress
                        | [ Label ] ( "DC" Address | "DS" KnownAddress )
                        | "ORG" KnownAddress | "BEG" .
    Label             = identifier .
    KnownAddress      = Address .
END ASM.
```

When it comes to developing a practical assembler, the first of these grammars appears to have the advantage of simplicity so far as syntax analysis is concerned - but this simplicity comes at a price, in that the static semantic constrainer would have to expend effort in distinguishing the various statement forms from one another. An assembler based on the second grammar would not leave so much to the semantic constrainer, but would apparently require a more complex parser. In later sections, using the simpler description as the basis of a parser, we shall see how both it and the constrainer are capable of development in an *ad hoc* way.

Neither of the above syntactic descriptions illustrates some of the pragmatic features that may beset a programmer using the ASSEMBLER language. Typical of these are restrictions or relaxations on case-sensitivity of identifiers, or constraints that labels may have to appear immediately at the start of a line, or that identifiers may not have more than a limited number of significant characters. Nor, unfortunately, can the syntactic description enforce some essential static semantic constraints, such as the requirement that each alphanumeric symbol used as an address should also occur uniquely in a label field of an instruction, or that the values of the address fields that appear with directives like `DS` and `ORG` must have been defined before the corresponding directives are first encountered. The description may *appear* to enforce these so-called *context-sensitive* features of the language, because the non-terminals have been given suggestive names like `KnownAddress`, but it turns out that a simple parser will not be able to enforce them on its own.

As it happens, neither of these grammars yet provides an adequate description for a compiler generator like Coco/R, for reasons that will become apparent after studying Chapter 9. The modifications needed for driving Coco/R may be left as an interesting exercise when the reader has had more experience in parsing techniques.

6.2 One- and two-pass assemblers, and symbol tables

Readers who care to try the assembly translation process for themselves will realize that this cannot easily be done on a single pass through the ASSEMBLER source code. In the example given earlier, the instruction

```
        BCC  EVEN
```

cannot be translated completely until one knows the address of `EVEN`, which is only revealed when the statement

```
EVEN    BNZ  LOOP
```

is encountered. In general the process of assembly is always non-trivial, the complication arising - even with programs as simple as this one - from the inevitable presence of **forward references**.

An assembler may solve these problems by performing two distinct passes over the user program. The primary aim of the first pass of a **two-pass assembler** is to draw up a **symbol table**. Once the first pass has been completed, all necessary information on each user defined identifier should have been recorded in this table. A second pass over the program then allows full assembly to take place quite easily, referring to the symbol table whenever it is necessary to determine an address for a named label, or the value of a named constant.

The first pass can perform other manipulations as well, such as some error checking. The second pass depends on being able to rescan the program, and so the first pass usually makes a copy of this on some backing store, usually in a slightly altered form from the original.

The behaviour of a two-pass assembler is summarized in Figure 6.1.

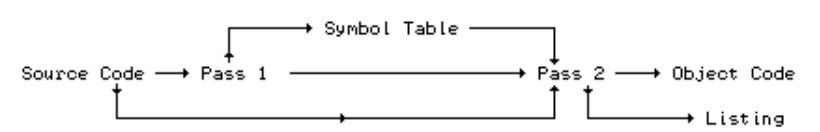


Figure 6.1 Flow of information through a two-pass assembler

The other method of assembly is via a **one-pass assembler**. Here the source is scanned but once, and the construction of the symbol table is rather more complicated, since outstanding references must be recorded for later *fixup* or *backpatching* once the appropriate addresses or values are revealed. In a sense, a two-pass assembler may be thought of as making two passes over the source program, while a one-pass assembler makes a single pass over the source program, followed by a later partial pass over the object program.

As will become clear, construction of a sophisticated assembler, using either approach, calls for a fair amount of ingenuity. In what follows we shall illustrate several principles rather simply and naïvely, and leave the refinements to the interested reader in the form of exercises.

Assemblers all make considerable use of tables. There are always (conceptually at least) two of these:

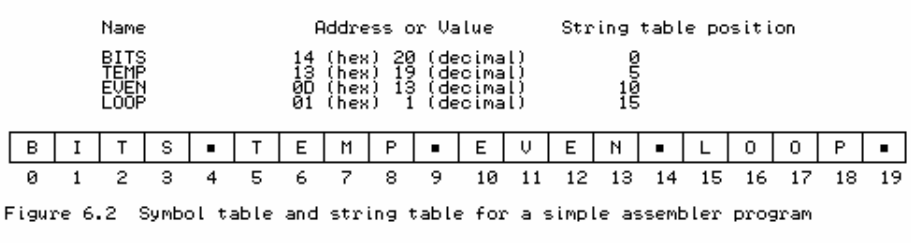
- The *Opcode Translation Table*. In this will be found matching pairs of mnemonics and their numerical equivalents. This table is of fixed length in simple assemblers.

- The *Symbol Table*. In this will be entered the user defined identifiers, and their corresponding addresses or values. This table varies in length with the program being assembled.

Two other commonly found tables are:

- The *Directive Table*. In this will be found mnemonics for the directives or pseudo-operations. The table is of fixed length, and is usually incorporated into the opcode translation table in simple assemblers.

- The *String Table*. As a space saving measure, the various user-defined names are often gathered into one closely packed table - effectively being stored in one long string, with some distinctive separator such as a `NUL` character between each sub-string. Each identifier in the symbol table is then cross-linked to this table. For example, for the program given earlier we might have a symbol table and string table as shown in Figure 6.2.


Name
Address or Value
String table position
BITS
14 (hex) 20 (decimal)
0
TEMP
13 (hex) 19 (decimal)
5
EVEN
0D (hex) 13 (decimal)
10
LOOP
01 (hex) 1 (decimal)
15
B I T S ▪ T E M P ▪ E V E N ▪ L O O P ▪
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Figure 6.2 Symbol table and string table for a simple assembler program

More sophisticated macro-assemblers need several other tables, so as to be able to handle user-defined opcodes, their parameters, and the source text which constitutes the definition of each macro. We return to a consideration of this point in the next chapter.

The first pass, as has been stated, has as its primary aim the creation of a symbol table. The "name" entries in this are easily made as the label fields of the source are read. In order to be able to complete the "address" entries, the first pass has to keep track, as it scans the source, of the so-called **location counter** - that is, the address at which each code and data value will later be located (when the code generation takes place). Such addresses are controlled by the directives `ORG` and `DS` (which affect the location counter explicitly), as well as by the directive `DC`, and, of course, by the opcodes which will later result in the creation of one or two machine words. The directive `EQU` is a special case; it simply gives a naming facility.

Besides constructing the symbol table, this pass must supervise source handling, and lexical, syntactic and semantic analysis. In essence it might be described by something on the lines of the following, where, we hasten to add, considerable liberties have been taken with the pseudo-code used to express the algorithm.

```
Initialize tables, and set Assembling := TRUE; Location := 0;
WHILE Assembling DO
  Read line of source and unpack into constituent fields
     Label, Mnemonic, AddressField (* which could be a Name or Number *)
  Use Mnemonic to identify Opcode from OpTable
  Copy line of source to work file for later use by pass two
  CASE Mnemonic OF
    "BEG" : Location := 0
    "ORG" : Location := AddressField.Number
    "DS " : IF Line.Labelled THEN SymbolTable.Enter(Label, Location)
            Location := Location + AddressField.Number
    "EQU" : SymbolTable.Enter(Label, AddressField.Number)
    "END" : Assembling := FALSE
    all others (* including DC *):
            IF Line.Labelled THEN SymbolTable.Enter(Label, Location)
            Location := Location + number of bytes to be generated
  END
```

```
END
```

The second pass is responsible mainly for code generation, and may have to repeat some of the source handling and syntactic analysis.

```
Rewind work file, and set Assembling := TRUE
WHILE Assembling DO
  Read a line from work file and unpack Mnemonic, Opcode, AddressField
  CASE Mnemonic OF
    "BEG" : Location := 0
    "ORG" : Location := AddressField.Number
    "DS " : Location := Location + AddressField.Number
    "EQU" : no action (* EQU dealt with on pass one *)
    "END" : Assembling := FALSE
    "DC " : Mem[Location] := ValueOf(AddressField); INC(Location)
    all others:
            Mem[Location] := Opcode;  INC(Location)
            IF two-byte Opcode THEN
              Mem[Location] := ValueOf(AddressField); INC(Location)
            END
  END
  Produce source listing of this line
END
```

6.3 Towards the construction of an assembler

The ideas behind assembly may be made clearer by slowly refining a simple assembler for the language given earlier, allowing only for the creation of fixed address, as opposed to relocatable code. We shall assume that the assembler and the assembled code can co-reside in memory. We are confined to write a cross-assembler, not only because no such real machine exists, but also because the machine is far too rudimentary to support a resident assembler - let alone a large C++ or Modula-2 compiler.

In C++ we can define a general interface to the assembler by introducing a class with a public interface on the lines of the following:

```
class AS {
  public:
    void assemble(bool &errors);
    // Assembles and lists program.
    // Assembled code is dumped to file for later interpretation, and left
    // in pseudo-machine memory for immediate interpretation if desired.
    // Returns errors = true if assembly fails

    AS(char *sourcename, char *listname, char *version, MC *M);
    // Instantiates version of the assembler to process sourcename, creating
    // listings in listname, and generating code for associated machine M
};
```

This public interface allows for the development of a variety of assemblers (simple, sophisticated, single-pass or multi-pass). Of course there are private members too, and these will vary somewhat depending on the techniques used to build the assembler. The constructor for the class creates a link to an instance of a machine class `MC` - we are aiming at the construction of an assembler for our hypothetical single-accumulator machine that will leave assembled code in the pseudo-machine's memory, where it can be interpreted as we have already discussed in Chapter 4. The main program for our system will essentially be developed on the lines of the following code:

```
void main(int argc, char *argv[])
{ bool errors;
  char SourceName[256], ListName[256];

  // handle command line parameters
  strcpy(SourceName, argv[1]);
  if (argc > 2) strcpy(ListName, argv[2]);
  else appendextension(SourceName, ".lst", ListName);
```

```
  // instantiate assembler components
  MC *Machine   = new MC();
  AS *Assembler = new AS(SourceName, ListName, "Assembler version 1", Machine);

  // start assembly
  Assembler->assemble(errors);

  // examine outcome and interpret if possible
  if (errors) { printf("\nAssembly failed\n"); }
  else { printf("\nAssembly successful\n"); Machine->interpret(); }
  delete Machine;
  delete Assembler;
}
```

This driver routine has made provision for extracting the file names for the source and listing files from command line parameters set up when the assembler program is invoked.

In using a language like C++ or Modula-2 to implement the assembler (or rather assemblers, since we shall develop both one-pass and two-pass versions of the assembler class), it is convenient to create classes or modules to be responsible for each of the main phases of the assembly process. In keeping with our earlier discussion we shall develop a source handler, scanner, and simple parser. In a two-pass assembler the parser is called from a first pass that follows parsing with static semantic analysis; control then passes to the second pass that completes code generation. In a one-pass assembler the parser is called in combination with semantic analysis and code generation.

On the source diskette that accompanies this book can be found a great deal of code illustrating this development, and the reader is urged to study this as he or she reads the text, since there is too much code to justify printing it all in this chapter. Appendix D contains a complete listing of the source code for the assembler as finally developed by the end of the next chapter.

6.3.1 Source handling

In terms of the overall translator structure illustrated in Figure 2.4, the first phase of an assembler will embrace the source character handler, which scans the source text, and analyses it into lines, from which the scanner will be then able to extract tokens or symbols. The public interface to a class for handling this phase might be:

```
class SH {
  public:
    FILE *lst; // listing file
    char ch;   // latest character read

    void nextch(void);
    // Returns ch as the next character on current source line, reading a new
    // line where necessary.  ch is returned as NUL if src is exhausted

    bool endline(void);
    // Returns true when end of current line has been reached

    bool startline(void);
    // Returns true if current ch is the first on a line

    void writehex(int i, int n);
    // Writes (byte valued) i to lst file as hex pair, left-justified in n spaces

    void writetext(char *s, int n);
    // Writes s to lst file, left-justified in n spaces

    SH();
    // Default constructor

    SH(char *sourcename, char *listname, char *version);
    // Opens src and lst files using given names
    // Initializes source handler, and displays version information on lst file

    ~SH();
    // Closes src and lst files
};
```

Some aspects of this interface deserve further comment:

- It is probably bad practice to declare variables like `ch` as public, as this leaves them open to external abuse. However, we have compromised here in the interests of efficiency.

- Client routines (like those which call `nextch`) should not have to worry about anything other than the values provided by `ch`, `startline()` and `endline()`. The main client routine is, of course, the lexical analyser.

- Little provision has been made here for producing a source listing, other than to export the file on which the listing might be made, and the mechanism for writing some version information and hexadecimal values to this file. A source line might be listed immediately it is read, but in the case of a two-pass assembler the listing is usually delayed until the second pass, when it can be made more complete and useful to the user. Furthermore, a free-format input can be converted to a fixed-format output, which will probably look considerably better.

The implementation of this class is straightforward and can be studied in Appendix D. As with the interface, some aspects of the implementation call for comment:

- `nextch` has to provide for situations in which it might be called after the input file has been exhausted. This situation should only arise with erroneous source programs, of course.

- Internally the module stores the source on a line-buffered basis, and adds a blank character to the end of each `line` (or a `NUL` character in the case where the source has ended). This is useful for ensuring that a symbol that extends to the end of a line can easily be recognized.

Exercises

6.1 A source handler implemented in this way will be found to be very slow on many systems, where each call to a routine to read a single character may involve a call to an underlying operating system. Experiment with the idea that the source handler first reads the entire source into a large memory buffer in one fell swoop, and then returns characters by extracting them from this buffer. Since memory (even on microcomputers) now tends to be measured in megabytes, while source programs are rather small, this idea is usually quite feasible. Furthermore, this suggestion overcomes the problem of using a line buffer of restricted size, as is used in our simple implementation.

6.3.2 Lexical analysis

The next phase to be tackled is that of lexical analysis. For our simple ASSEMBLER language we recognize immediately that source characters can only be assembled into numbers, alphanumeric names (as for labels or opcodes) or comment strings. Accordingly we adopt the following public interface to our scanner class:

```
enum LA_symtypes {
  LA_unknown, LA_eofsym, LA_eolsym, LA_idsym, LA_numsym, LA_comsym
};


struct LA_symbols {
  bool islabel;           // if in first column
  LA_symtypes sym;        // class
```

```
  ASM_strings str;        // lexeme
  int num;                // value if numeric
};


class LA {
  public:
    void getsym(LA_symbols &SYM, bool &errors);
    // Returns the next symbol on current source line.
    // Sets errors if necessary and returns SYM.sym = unknown if no
    // valid symbol can be recognized

    LA(SH *S);
    // Associates scanner with its source handler S
};
```

where we draw the reader's attention to the following points:

- The `LA_symbols` structure allows the client to recognize that the first symbol found on a line has defined a label if it began in the very first column of the line - a rather messy feature of our ASSEMBLER language.

- In ASSEMBLER programs, the ends of lines become significant (which is not the case with languages like C++, Pascal or Modula-2), so that it is useful to introduce `LA_eolsym` as a possible symbol type.

- Similarly, we must make provision for not being able to recognize a symbol (by returning `LA_unknown`), or not finding a symbol (`LA_eofsym`).

Developing the `getsym` routine for the recognition of these symbols is quite easy. It is governed essentially by the lexical grammar (defined in the `TOKENS` section of our Cocol specification given earlier), and is sensibly driven by a `switch` or `CASE` statement that depends on the first character of the token. The essence of this - again taking considerable liberties with syntax - may be expressed

```
BEGIN
  skip leading spaces, or to end of line
  recognize end-of-line and start-of-line conditions, else
  CASE CH OF
    letters: SYM.Sym := LA_idsym;  unpack word;
    digits : SYM.Sym := LA_numsym; unpack number;
    ';'    : SYM.Sym := LA_comsym; unpack comment;
    ELSE   : SYM.Sym := LA_unknown
  END
END
```

A detailed implementation may be found on the source diskette. It is worth pointing out the following:

- All fields (attributes) of `SYM` are well defined after a call to `getsym`, even those of no immediate interest.

- While determining the value of `SYM.num` we also copy the digits into `SYM.name` for the purposes of later listing. At this stage we have assumed that overflow will not occur in the computation of `SYM.num`.

- Identifiers and comments that are too long are ruthlessly truncated.

- Identifiers are converted to upper case for consistency. Comments are preserved unchanged.

Exercises

6.2 First extend the lexical grammar, and then extend the lexical analyser to allow hexadecimal constants as alternatives in addresses, for example

```
          LAB  LDI  $0A        ; 0A(hex) = 10(decimal)
```

6.3 Another convention is to allow hexadecimal constants like 0FFh or 0FFH, with the trailing H implying hexadecimal. A hex number must, however, start with a digit in the range '0' .. '9', so that it can be distinguished from an identifier. Extend the lexical grammar, and then implement this option. Why is it harder to handle than the convention suggested in Exercise 6.2?

6.4 Extend the grammar and the analyser to allow a single character as an operand or address, for example

```
          LAB    LDI  'A'        ; load immediate 'A' (ASCII 041H)
```

The character must, of course, be converted into the corresponding ordinal value by the assembler. How can one allow the quote character itself to be used as an address?

6.5 If the entire source of the program were to be read into memory as suggested in Exercise 6.1 it would no longer be necessary to copy the `name` field for each symbol. Instead, one could use two numeric fields to record the starting position and the length of each name. Modify the lexical analyser to use such a technique. Clearly this will impact the detailed implementation of some later phases of assembly as well - see Exercise 6.8.

6.6 As an alternative to storing the entire source program in memory, explore the possibility of constructing a string table on the lines of that discussed in section 6.2.

6.3.3 Syntax analysis

Our suggested method of syntax analysis requires that each free format source line be decomposed in a consistent way. A suitable public interface for a simple class that handles this phase is given below:

```
enum SA_addresskinds { SA_absent, SA_numeric, SA_alphameric };

struct SA_addresses {
  SA_addresskinds kind;
  int number;     // value if known
  ASM_alfa name;  // character representation
};

struct SA_unpackedlines {
  // source text, unpacked into fields
  bool labelled, errors;
  ASM_alfa labfield, mnemonic;
  SA_addresses address;
  ASM_strings comment;
};

class SA {
  public:
    void parse(SA_unpackedlines &srcline);
    // Analyses the next source line into constituent fields

    SA(LA * L);
    // Associates syntax analyser with its lexical analyser L
};
```

and, as before, some aspects of this deserve further comment:

- The `SA_addresses` structure has been introduced to allow for later extensibility.

- The `SA_unpackedlines` structure makes provision for recording whether a source line has been labelled. It also makes provision for recording that the line is erroneous. Some errors might be detected when the syntax analysis is performed; others might only be detected when the constraint analysis or code generation are attempted.

- Not only does syntax analysis in the first pass of a two-pass assembler require that we unpack a source line into its constituent fields, using the `getsym` routine, the first pass also has to be able to write the source line information to a work file for later use by the second pass. It is convenient to do this *after* unpacking, to save the necessity of re-parsing the source on the second pass.

The routine for unpacking a source line is relatively straightforward, but has to allow for various combinations of present or absent fields. The syntax analyser can be programmed by following the EBNF productions given in Cocol under the `PRODUCTIONS` section of the simpler grammar in section 6.1, and the implementation on the source diskette is worthy of close study, bearing in mind the following points:

- The analysis is very *ad hoc*. This is partly because it has to take into account the possibility of errors in the source. Later in the text we shall look at syntax analysis from a rather more systematic perspective, but it is usually true that syntax analysers incorporate various messy devices for side-stepping errors.

- Every field is well defined when analysis is complete - default values are inserted where they are not physically present in the source.

- Should the source text become exhausted, the syntax analyser performs "error correction", effectively by creating a line consisting only of an `END` directive.

- When an unrecognizable symbol is detected by the scanner, the syntax analyser reacts by recording that the line is in error, and then copies the rest of the line to the `comment` field. In this way it is still possible to list the offending line in some form at a later stage.

- The simple routine for `getaddress` will later be modified to allow expressions as addresses.

Exercises

6.7 At present mnemonics and user defined identifiers are both handled in the same way. Perhaps a stronger distinction should be drawn between the two. Then again, perhaps one should allow mnemonics to appear in address fields, so that an instruction like

```
LAB   LDI   LDI     ;  A := 27
```

would become legal. What modifications to the underlying grammar and to the syntax analyser would be needed to implement any ideas you may have on these issues?

6.8 How would the syntax analyser have to be modified if we were to adopt the suggestion that all the source code be retained in memory during the assembly process? Would it be necessary to unpack each line at all?

6.3.4 The symbol table interface

We define a clean public interface to a symbol table handler, thus allowing us to implement various strategies for symbol table construction without disturbing the rest of the system. The interface chosen is

```
typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v);

class ST {
  public:
    void printsymboltable(bool &errors);
    // Summarizes symbol table at end of assembly, and alters errors
    // to true if any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes mem[], ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)
};
```

6.4 Two-pass assembly

For the moment we shall focus attention on a two-pass assembler, and refine the code from the simple algorithms given earlier. The first pass is mainly concerned with static semantics, and with constructing a symbol table. To be able to do this, it needs to keep track of a location counter, which is updated as opcodes are recognized, and which may be explicitly altered by the directives `ORG`, `DS` and `DC`.

6.4.1 Symbol table structures

A simple implementation of the symbol table handler outlined in the last section, suited to two-pass assembly, is to be found on the source diskette. It uses a dynamically allocated stack, in a form that should readily be familiar to students of elementary data structures. More sophisticated table handlers usually employ a so-called **hash table**, and are the subject of later discussion. The reader should note the following:

- For a two-pass assembler, labels are entered into the table (by making calls on `enter`) only when their *defining occurrences* are encountered during the first pass.
- On the second pass, calls to `valueofsymbol` will be made when *applied occurrences* of labels are encountered.
- For a two-pass assembler, function type `ST_patch` and function `outstandingreferences` are irrelevant - as, indeed, is the `location` parameter to `valueofsymbol`.
- The symbol table entries are very simple structures defined by

```
struct ST_entries {
  ASM_alfa name;      // name
```

```
  MC_bytes value;     // value once defined
  bool defined;       // true after defining occurrence encountered
  ST_entries *slink;  // to next entry
};
```

6.4.2 The first pass - static semantic analysis

Even though no code is generated until the second pass, the location counter (marking the address of each byte of code that is to be generated) must be tracked on both passes. To this end it is convenient to introduce the concept of a *code line* - a partial translation of each *source line*. The fields in this structure keep track of the location counter, opcode value, and address value (for two-byte instructions), and are easily assigned values after extending the analysis already performed by the syntax analyser. These extensions effectively constitute static semantic analysis. For each unpacked source line the analysis is required to examine the `mnemonic` field and - if present - to attempt to convert this to an opcode, or to a directive, as appropriate. The `opcode` value is then used as the despatcher in a switching construct that keeps track of the location counter and creates appropriate entries in the symbol table whenever defining occurrences of labels are met.

The actual code for the first pass can be found on the source diskette, and essentially follows the basic algorithm outlined in section 6.2. The following points are worth noting:

- Conversion from `mnemonic` to `opcode` requires the use of some form of opcode table. In this implementation we have chosen to construct a table that incorporates both the machine opcodes and the directive pseudo-opcodes in one simple sorted list, allowing a simple binary search to locate a possible opcode entry quickly.

  An alternative strategy might be to incorporate the opcode table into the scanner, and to handle the conversion as part of the syntax analysis, but we have chosen to leave that as the subject of an exercise.

- The attempt to convert a mnemonic may fail in two situations. In the case of a line with a blank opcode field we may sensibly return a fictitious legal empty opcode. However, when an opcode is present, but cannot be recognized (and must thus be assumed to be in error) we return a fictitious illegal opcode `err`.

- The system makes use of an intermediate work file for communicating between the two passes. This file can be discarded after assembly has been completed, and so can, in principle, remain hidden from the user.

- The arithmetic on the location counter `location` must be done *modulo* 256 because of the limitations of the target machine.

- Our assembler effectively requires that all identifiers used as labels must be "declared". In this context this means that all the identifiers in the symbol table must have appeared in the label field of some source line, and should all have been entered into the symbol table by the end of the first pass. When appropriate, we determine the value of an address, either directly, or from the symbol table, by calling the table handler routine `valueofsymbol`, which returns a parameter indicating the success of the search. It might be thought that failure is ruled out, and that calls to this routine are made only in the second pass. However, source lines using the directives `EQU, DS` and `ORG` may have address fields specified in terms of labels, and so even on the first pass the assembler may have to refer to the values of these labels. Clearly chaos will arise if the labels used in the address fields for these directives are not declared before use, and the assembler must be prepared to flag violations of this principle as errors.

6.4.3 The second pass - code generation

The second pass rescans the program by extracting partially assembled lines from the intermediate file, and then passing each of these to the code generator. The code generator has to keep track of further errors that might arise if any labels were not properly defined on the first pass. Because of the work already done in the first pass, handling the directives is now almost trivial in this pass.

Once again, complete code for a simple implementation is to be found on the source diskette, and it should be necessary only to draw attention to the following points:

- For our simple machine, all the generated objected code can be contained in an array of length 256. A more realistic assembler might not be able to contain the entire object code in memory, because of lack of space. For a two-pass assembler few problems would arise, as the code could be written out to a file as soon as it was generated.

- Exactly how the object code is finally to be treated is a matter for debate. Here we have called on the `listcode` routine from the class defining the pseudo-machine, which dumps the 256 bytes in a form that is suitable for input to a simple loader. However, the driver program suggested earlier also allows this code to be interpreted immediately after assembly has been successful.

- An assembler program typically gives the user a listing of the source code, usually with assembled code alongside it. Occasionally extra frills are provided, like cross reference tables for identifiers and so on. Our one is quite simple, and an example of a source listing produced by this assembler was given earlier.

Exercises

6.9 Make an extension to the ASSEMBLER language, to its grammar, and to the assembler program, to allow a character string as an operand in the `DC` directive. For example

```
TYRANT  DC  "TERRY"
```

should be treated as equivalent to

```
TYRANT  DC  'T'
        DC  'E'
        DC  'R'
        DC  'R'
        DC  'Y'
```

Is it desirable or necessary to delimit strings with different quotes from those used by single characters?

6.10 Change the table handler so that the symbol table is stored in a binary search tree, for efficiency.

6.11 The assembler will accept a line consisting only of a non-empty `LABEL` field. Is there any advantage in being able to do this?

6.12 What would happen if a label were to be *defined* more than once?

6.13 What would happen if a label were left undefined by the end of the first pass?

6.14 How would the symbol table handling alter if the source code were all held in memory throughout assembly (see Exercise 6.1), or if a string table were used (see Exercise 6.6)?

6.5 One-pass assembly

As we have already mentioned, the main reason for having the second pass is to handle the problem of *forward references* - that is, the use of labels before their locations or values have been defined. Most of the work of lexical analysis and assembly can be accomplished directly on the first pass, as can be seen from a close study of the algorithms given earlier and the complete code used for their implementation.

6.5.1 Symbol table structures

Although a one-pass assembler not always be able to determine the value of an address field immediately it is encountered, it is relatively easy to cope with the problem of forward references. We create an additional field `flink` in the symbol table entries, which then take the form

```
struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool defined;           // true after defining occurrence encountered
  ST_entries *slink;      // to next entry
  ST_forwardrefs *flink;  // to forward references
};
```

The `flink` field points to entries in a **forward reference table**, which is maintained as a set of linked lists, with nodes defined by

```
struct ST_forwardrefs {   // forward references for undefined labels
  MC_bytes byte;          // to be patched
  ST_forwardrefs *nlink;  // to next reference
};
```

The `byte` fields of the `ST_forwardrefs` nodes record the addresses of as yet incompletely defined object code bytes.

6.5.2 The first pass - analysis and code generation

When reference is made to a label in the address field of an instruction, the `valueofsymbol` routine searches the symbol table for the appropriate entry, as before. Several possibilities arise:

- If the label has already been defined, it will already be in the symbol table, marked as `defined = true`, and the corresponding address or value can immediately be obtained from the `value` field.

- If the label is not yet in the symbol table, an entry is made in this table, marked as `defined = false`. The `flink` field is then initialized to point to a newly created entry in the forward reference table, in the `byte` field of which is recorded the address of the object byte whose value has still to be determined.

- If the label is already in the symbol table, but still flagged as `defined = false`, then a further entry is made in the forward reference table, linked to the earlier entries for this label.

This may be made clearer by considering the same program as before (shown fully assembled, for convenience).

```
00                  BEG                 ; count the bits in a number
00    0A            INI                 ; Read(A)
01          LOOP                        ; REPEAT
01    16            SHR                 ;  A := A DIV 2
02    3A 0D         BCC     EVEN        ;  IF A MOD 2 # 0 THEN
04    1E 13         STA     TEMP        ;    TEMP := A
06    19 14         LDA     BITS
08    05            INC
09    1E 14         STA     BITS        ;    BITS := BITS + 1
0B    19 13         LDA     TEMP        ;    A := TEMP
0D    37 01 EVEN    BNZ     LOOP        ; UNTIL A = 0
0F    19 14         LDA     BITS        ;
11    0E            OTI                 ; Write(BITS)
12    18            HLT                 ; terminate execution
13          TEMP    DS      1           ; VAR TEMP : BYTE
14    00    BITS    DC      0           ;     BITS : BYTE
15                  END
```

When the instruction at 02h (`BCC EVEN`) is encountered, `EVEN` is entered in the symbol table, undefined, linked to an entry in the forward reference table, which refers to 03h. Assembly of the next instruction enters `TEMP` in the symbol table, undefined, linked to a new entry in the forward reference table, which refers to 05h. The next instruction adds `BITS` to the symbol table, and when the instruction at 09h (`STA BITS`) is encountered, another entry is made to the forward reference table, which refers to 0Ah, itself linked to the entry which refers to 07h. This continues in the same vein, until by the time the instruction at 0Dh (`EVEN BNZ LOOP`) is encountered, the tables are as shown in Figure 6.3.

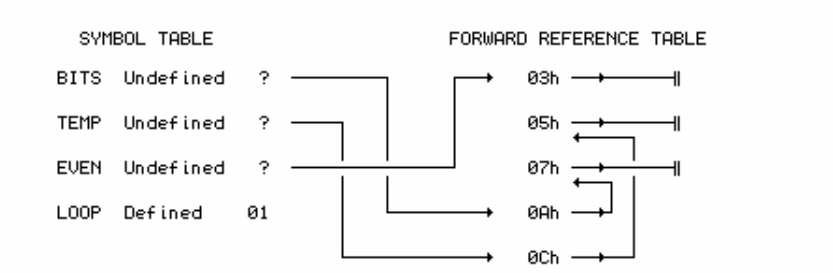


Figure 6.3 Symbol table and forward references part way through assembly

In passing, we might comment that in a real system this strategy might lead to extremely large structures. These can, fairly obviously, be kept smaller if the bytes labelled by the `DC` and `DS` instructions are all placed before the "code" which manipulates them, and some assemblers might even insist that this be done.

Since we shall also have to examine the symbol table whenever a label is defined by virtue of its appearance in the label field of an instruction or directive, it turns out to be convenient to introduce a private routine `findentry`, internal to the table handler, to perform the symbol table searching.

```
void findentry(ST_entries *&symentry, char *name, bool &found);
```

This involves a simple algorithm to scan through the symbol table, being prepared for either finding or not finding an entry. In fact, we go further and code the routine so that it *always* finds an appropriate entry, if necessary creating a new node for the purpose. Thus, `findentry` is a routine with side-effects, and so might be frowned upon by the purists. The parameter `found` records whether the entry refers to a previously created node or not.

The code for `enter` also changes somewhat. As already mentioned, when a non-blank `label` field

is encountered, the symbol table is searched. Two possibilities arise:

- If the `label` was not previously there, the new entry is completed, flagged as `defined = true`, and its `value` field is set to the now known value.
- If it was previously there, but flagged `defined = false`, the extant symbol table entry is updated, with `defined` set to `true`, and its `value` field set to the now known value.

At the end of assembly the symbol table will, in most situations, contain entries in the forward reference lists. Our table handler exports an `outstandingreferences` routine to allow the assembler to walk through these lists. Rather than have the symbol table handler interact directly with the code generation process, this pass is accomplished by applying a procedural parameter as each node of the forward reference lists is visited. In effect, rather than making a second pass over the source of the program, a partial second pass is made over the object code.

This may be made clearer by considering the same program fragment as before. When the definition of `BITS` is finally encountered at 14h, the symbol table and forward reference table will effectively have become as shown in Figure 6.4.

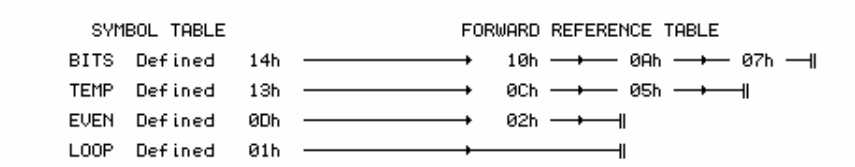


Figure 6.4 Symbol table and forward references at the end of assembly

---

Exercises

6.15 What modifications (if any) are needed to incorporate the extensions suggested as exercises at the end of the last section into the simple one-pass assembler?

6.16 We mentioned in section 6.4.3 that there was no great difficulty in assembling large programs with a two-pass assembler. How does one handle programs too large to co-reside with a one-pass assembler?

6.17 What currently happens in our one-pass assembler if a label is redefined? Should one be allowed to do this (that is, is there any advantage to be gained from being allowed to do so), and if not, what should be done to prevent it?

6.18 Constructing the forward reference table as a dynamic linked structure may be a little grander than it needs to be. Explore the possibility of holding the forward reference chains within the code being assembled. For example, if we allow the symbol table entries to be defined as follows

```
struct ST_entries {
  ASM_alfa name;      // name
  MC_bytes value;     // value once defined
  bool defined;       // true after defining occurrence encountered
  ST_entries *slink;  // to next entry
  MC_bytes flink;     // to forward references
};
```

we can arrange that the latest forward reference "points" to a byte in memory in which will be

stored the label's `value` once this is known. In the interim, this byte contains a pointer to an earlier byte in memory where the same `value` has ultimately to be recorded. For the same program fragment as was used in earlier illustrations, the code would be stored as follows, immediately before the final `END` directive is encountered. Within this code the reader should note the chain of values 0Ah, 07h, 00h (the last of which marks the end of the list) giving the list of bytes where the value of `BITS` is yet to be stored.

```
00                  BEG          ; count the bits in a number
00  0A              INI          ; read(A)
01          LOOP                 ; REPEAT
01  16              SHR          ;  A := A DIV 2
02  3A 00           BCC EVEN     ;  IF A MOD 2 # 0 THEN
04  1E 00           STA TEMP     ;    TEMP := A
06  19 00           LDA BITS
08  05              INC
09  1E 07           STA BITS     ;    BITS := BITS + 1
0B  19 05           LDA TEMP     ;    A := TEMP
0D  37 01  EVEN     BNZ LOOP     ; UNTIL A = 0
0F  19 0A           LDA BITS     ;
11  0E              OTI          ; Write(BITS)
12  18              HLT          ; terminate execution
13          TEMP    DS  1        ; VAR TEMP : BYTE
14  00      BITS    DC  0        ;     BITS : BYTE
15                  END
```

By the end of assembly the forward reference table effectively becomes as shown below. The outstanding references may be fixed up in much the same way as before, of course.

```
Name    Defined    Value   FLink

BITS     true       14h     0Ah
TEMP     true       13h     0Ch
EVEN     true       0Dh     00h
LOOP     true       01h     00h
```

7 ADVANCED ASSEMBLER FEATURES

It cannot be claimed that the assemblers of the last chapter are anything other than toys - but by now the reader will be familiar with the drawbacks of academic courses. In this chapter we discuss some extensions to the ideas put forward previously, and then leave the reader with a number of suggestions for exercises that will help turn the assembler into something more closely resembling the real thing.

Complete source code for the assembler discussed in this chapter can be found in Appendix D. This source code and equivalent implementations in Modula-2 and Pascal are also to be found on the accompanying source diskette.

7.1 Error detection

Our simple assemblers are deficient in a very important area - little attempt is made to report errors in the source code in a helpful manner. As has often been remarked, it is very easy to write a translator if one can assume that it will only be given correctly formed programs. And, as the reader will soon come to appreciate, error handling adds considerably to the complexity of any translation process.

Errors can be classified on the basis of the stage at which they can be detected. Among others, some important potential errors are as follows:

Errors that can be detected by the source handler

- Premature end of source file - this might be a rather fatal error, or its detection might be used to supply an effective `END` line, as is done by some assemblers, including our own.

Errors that can be detected by the lexical analyser

- Use of totally unrecognizable characters.
- Use of symbols whose names are too long.
- Comment fields that are too wide.
- Overflow in forming numeric constants.
- Use of non-digit characters in numeric literals.
- Use of symbols in the label field that do not start with a letter.

Errors that can be detected by the syntax analyser

- Use of totally unrecognizable symbols, or misplaced symbols, such as numbers where the comment field should appear.

- Failure to form address fields correctly, by misplacing operators, omitting commas in parameter lists, and so on.

Errors that can be detected by the semantic analyser

These are rather more subtle, for the semantics of ASSEMBLER programming are often deliberately vague. Some possible errors are:

- Use of undefined mnemonics.
- Failure to define all labels.
- Supplying address fields for one-byte instructions, or for directives like `BEG, END`.
- Omitting the address for a two-byte instruction, or for directives like `DS` or `DC`.
- Labelling any of the `BEG, ORG, IF` or `END` directives.
- Supplying a non-numeric address field to `ORG` or `EQU`. (This might be allowed in some circumstances).
- Attempting to reference an address outside the available memory. A simple recovery action here is to treat all addresses *modulo* the available memory size, but this, almost certainly, needs reporting.
- Use of the address of "data" as the address in a "branch" instruction. This is sometimes used in clever programming, and so is not usually regarded as an error.
- Duplicate definition, either of macro names, of formal parameter names, or of label names. This may allow trick effects, but should probably be discouraged.
- Failure to supply the correct number of actual parameters in a macro expansion.
- Attempting to use address fields for directives like `ORG, DS, IF` and `EQU` that cannot be fully evaluated at the time these directives take effect. This is a particularly nasty problem in a one-pass system, for forward references will be set up to object bytes that have no real existence.

The above list is not complete, and the reader is urged to reflect on what other errors might be made by the user of the assembler.

A moment’s thought will reveal that many errors can be detected during the first pass of a two-pass assembler, and it might be thought reasonable not to attempt the second pass if errors are detected on the first one. However, if a complete listing is to be produced, showing object code alongside source code, then this will have to wait for the second pass if forward references are to be filled in.

How best to report errors is a matter of taste. Many assemblers are fairly cryptic in this regard, reporting each error only by giving a code number or letter alongside the line in the listing where the error was detected. A better approach, exemplified in our code, makes use of the idea of constructing a *set* of errors. We then associate with each parsed line, not a Boolean error field, but one of some suitable set type. As errors are discovered this set can be augmented, and at an

appropriate time error reporting can take place using a routine like `listerrors` that can be found in the enhanced assembler class in Appendix D.

This is very easily handled with implementation languages like Modula-2 or Pascal, which directly support the idea of a set type. In C++ we can make use of a simple template set class, with operators overloaded so as to support virtually the same syntax as is found in the other languages. Code for such a class appears in the appendix.

7.2 Simple expressions as addresses

Many assemblers allow the programmer the facility of including expressions in the address field of instructions. For example, we might have the following (shown fully assembled, and with some deliberate quirks of coding):

```
Macro Assembler 1.0 on 30/05/96 at 21:47:53

(One Pass Assembler)

00                BEG             ; Count chars and lowercase letters
00        LOOP                    ; LOOP
00  0D            INA             ;   Read(CH)
01  2E 2E         CPI  PERIOD     ;   IF CH = "." THEN EXIT
03  36 19         BZE  EXIT
05  2E 61         CPI  SMALLZ - 25 ;   IF (CH >= "a")
07  39 12         BNG  * + 10
09  2E 7B         CPI  SMALLZ + 1  ;      AND (CH <= "z")
0B  38 12         BPZ  * + 6
0D  19 20         LDA  LETTERS     ;     THEN INC(Letters)
0F  05            INC
10  1E 20         STA  LETTERS     ;   END
12  19 21         LDA  LETTERS + 1 ;   INC(Total)
14  05            INC
15  1E 21         STA  LETTERS + 1
17  35 00         BRN  LOOP        ; END
19  19 20  EXIT   LDA  LETTERS
1B  0F            OTC              ; Write(Letters)
1C  19 21         LDA  TOTAL
1E  0F            OTC              ; Write(Total)
1F  18            HLT
20  00    LETTERS DC   0           ; RECORD Letters, Total : BYTE END
21        TOTAL   EQU  *
21  00            DC   0
22        SMALLZ  EQU  122         ; ascii 'z'
22        PERIOD  EQU  46          ; ascii '.'
22                END
```

Here we have used addresses like `LETTERS + 1` (meaning one location after that assigned to `LETTERS`), `SMALLZ-25` (meaning, in this case, an obvious 97), and `* + 6` and `* + 10` (a rather dangerous notation, meaning "6 bytes after the present one" and "10 bytes after the present one", respectively). These are typical of what is allowed in many commercial assemblers. Quite how complicated the expressions can become in a real assembler is not a matter for discussion here, but it is of interest to see how to extend our one-pass assembler if we restrict ourselves to addresses of a form described by

```
Address  =   Term { "+" Term  |  "-" Term } .
Term     =   Label  |  number |  "*" .
```

where `*` stands for "address of this byte". Note that we can, in principle, have as many terms as we like, although the example above used only one or two.

In a one-pass assembler, address fields of this kind can be handled fairly easily, even allowing for the problem of forward references. As we assemble each line we compute the value of each address

field as fully as we can. In some cases (as in `* + 6`) this will be completely; in other cases forward references will be needed. In the forward reference table entries we record not only the address of the bytes to be altered when the labels are finally defined, but also whether these values are later to be added to or subtracted from the value already residing in that byte. There is a slight complication in that all expressions must be computed *modulo* 256 (corresponding to a two's complement representation).

Perhaps this will be made clearer by considering how a one-pass assembler would handle the above code, where we have deliberately delayed the definition of `LETTERS, TOTAL, SMALLZ` and `PERIOD` till the end. For the `LETTERS + 1` address in instructions like `STA  LETTERS + 1` we assemble as though the instruction were `STA  1`, and for the `SMALLZ - 25` address in the instruction `CPI  SMALLZ - 25` we assemble as though the instruction were `CPI  -25`, or, since addresses are computed *modulo* 256, as though the instruction were `CPI 231`. At the point just before `LETTERS` is defined, the assembled code would look as follows:

```
Macro Assembler 1.0 on 30/05/96 at 21:47:53

(One Pass Assembler)

00                BEG             ; Count chars and lowercase letters
00         LOOP                   ; LOOP
00  0D             INA             ;   Read(CH)
01  2E 00          CPI  PERIOD     ;   IF CH = "." THEN EXIT
03  36 00          BZE  EXIT
05  2E E7          CPI  SMALLZ - 25 ;   IF (CH >= "a")
07  39 12          BNG  * + 10
09  2E 01          CPI  SMALLZ + 1  ;      AND (CH <= "z")
0B  38 12          BPZ  * + 6
0D  19 00          LDA  LETTERS     ;     THEN INC(Letters)
0F  05             INC
10  1E 00          STA  LETTERS     ;   END
12  19 01          LDA  LETTERS + 1 ;   INC(Total)
14  05             INC
15  1E 01          STA  LETTERS + 1
17  35 00          BRN  LOOP        ; END
19  19 00  EXIT    LDA  LETTERS
1B  0F             OTC              ; Write(Letters)
1C  19 00          LDA  TOTAL
1E  0F             OTC              ; Write(Total)
1F  18             HLT
20  00     LETTERS DC   0           ; RECORD Letters, Total : BYTE END
21         TOTAL   EQU  *
21  00             DC   0
22         SMALLZ  EQU  122         ; ascii 'z'
22         PERIOD  EQU  46          ; ascii '.'
22                 END
```

with the entries in the symbol and forward reference tables as depicted in Figure 7.1.

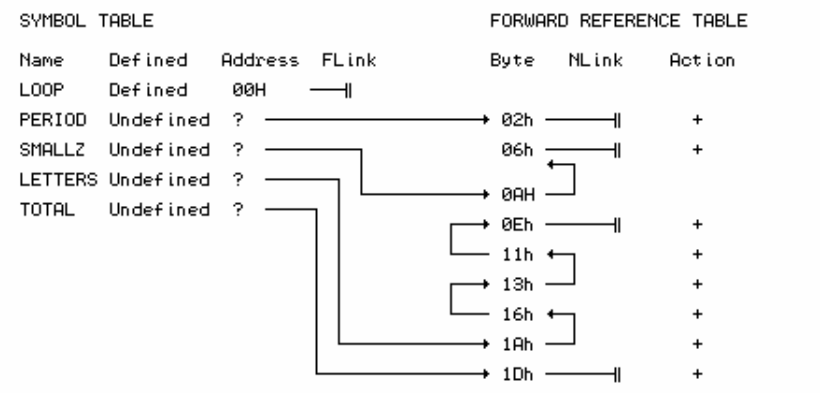


Figure 7.1 Symbol table and forward reference table when simple expressions are allowed to form composite address fields

To incorporate these changes requires modifications to the lexical analyser, (which now has to be able to recognize the characters +, - and * as corresponding to lexical tokens or symbols), to the syntax analyser (which now has more work to do in decoding the address field of an instruction - what was previously the complete address is now possibly just one term of a complex address), and to the semantic analyser (which now has to keep track of how far each address has been computed, as well as maintaining the symbol table).

Some of these changes are almost trivial: in the lexical analyser we simply extend the `LA_symtypes` enumeration, and modify the `getsym` routine to recognize the comma, plus, minus and asterisk as new tokens.

The changes to the syntax analyser are more profound. We change the definition of an unpacked line:

```
const int SA_maxterms = 16;

enum SA_termkinds {
  SA_absent, SA_numeric, SA_alphameric, SA_comma, SA_plus, SA_minus, SA_star
};

struct SA_terms {
  SA_termkinds kind;
  int number;      // value if known
  ASM_alfa name;   // character representation
};

struct SA_addresses {
  char length;     // number of fields
  SA_terms term[SA_maxterms - 1];
};

struct SA_unpackedlines {
  // source text, unpacked into fields
  bool labelled;
  ASM_alfa labfield, mnemonic;
  SA_addresses address;
  ASM_strings comment;
  ASM_errorset errors;
};
```

and provide a rather grander routine for doing the syntax analysis, which also takes more care to detect errors than before. Much of the spirit of this analysis is similar to the code used in the previous assemblers; the main changes occur in the `getaddress` routine. However, we should comment on the choice of an array to store the entries in an address field. Since each line will have a varying number of terms it might be thought better (especially with all the practice we have been having!) to use a dynamic structure. This has not been done here because we do not really need to create a new structure for each line - once we have assembled a line the address field is of no further interest, and the structure used to record it is thus reusable. However, we need to check that the capacity of the array is never exceeded.

The semantic actions needed result in a considerable extension to the algorithm used to evaluate an address field. The algorithm used previously is delegated to a `termvalue` routine, one that is called repeatedly from the main `evaluate` routine. The forward reference handling is also marginally more complex, since the forward reference entries have to record the outstanding action to be performed when the back-patching is finally attempted. The revised table handler interface needed to accommodate this is as follows:

```
enum ST_actions { ST_add, ST_subtract };

typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v, ST_actions a);

class ST {
  public:
    void printsymboltable(bool &errors);
```

```
    // Summarizes symbol table at end of assembly, and alters errors
    // to true if any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // Records action to be applied later in fixing up forward references.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes mem[], ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)
};
```

Exercises

7.1 Is it possible to allow a one-pass assembler to handle address fields that contain more general forms of expression, including multiplication and division? Attempt to do so, restricting your effort to the case where the expression is evaluated strictly from left to right.

7.2 One drawback of using dynamic structures for storing the elements of a composite address field is that it may be difficult to recover the storage when the structures are destroyed or are no longer needed. Would this drawback detract from their use in constructing the symbol table or forward reference table?

7.3 Improved symbol table handling - hash tables

In assembly, a great deal of time can be spent looking up identifiers and mnemonics in tables, and it is worthwhile considering how one might improve on the very simple linear search used in the symbol table handler of the previous chapter. A popular way of implementing very efficient table look-up is through the use of **hashing functions**. These are discussed at great length in most texts on data structures, and we shall provide only a very superficial discussion here, based on the idea of maintaining a symbol table in an array of fixed maximum length. For an assembler for a machine as simple as the one we are considering, a fairly small array would surely suffice. Although the possibilities for choosing identifiers are almost unlimited, the choice for any one program will be severely limited - after all, with only 256 bytes in the machine, we are scarcely likely to want to define even as many as 256 labels!

With this in mind we might set up a symbol table structure based on the following declarations:

```
struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool used;              // true after entry made in a table slot
  bool defined;           // true after defining occurrence encountered
  ST_forwardrefs *flink;  // to forward references
};


const int tablemax = 239; // symbol table size (prime number)
ST_entries hashtable[tablemax + 1];
```

The table is initialized by setting the `used` field for each entry to `false` before assembly commences; every time a new entry is made in the table this field is set to `true`.

The fundamental idea behind hashing is to define a simple function based on the characters in an identifier, and to use the returned value as an initial index or key into the table, at which position we hope to be able to store or find the identifier and its associated value. If we are lucky, all identifiers will map to rather scattered and different keys, making or finding an entry in the table will never take more than one comparison, and by the end of assembly there will still be unused slots in the table, and possibly large gaps between the slots that are used.

Of course, we shall never be totally lucky, except, perhaps, in trivial programs. Hash functions are kept very simple so that they can be computed quickly. The simplest of such functions will have the undesirable property that many different identifiers may map onto the same key, but a little reflection will show that, no matter how complicated one makes the function, one always runs the risk that this will happen. Some hash functions are clearly very risky - for example, simply using the value of the first letter in the identifier as a key. It would be much better to use something like

```
hash = (ident[first] * ident[last]) % tablemax;
```

(which would still fail to discriminate between identifiers like ABC and CBA), or

```
hash = (ident[first] * 256 + ident[last]) % tablemax;
```

(which would still fail to discriminate between identifiers like AC and ABC).

The subtle part of using a hash table concerns the action to take when we find that some other identifier is occupying the slot identified by the key (when we want to add to the table) or that a different identifier is occupying the slot (when we want to look up the value of an identifier in the table).

If this happens - an event known as a **collision** - we must be prepared to probe elsewhere in the table looking for the correct entry, a process known as **rehashing**. This can be done in a variety of ways. The easiest is simply to make a simple linear search in unit steps from the position identified by the key. This suffers from the disadvantage that the entries in the table tend to get very clustered - for example, if the key is simply the first letter, the first identifier starting with *A* will grab the obvious slot, the second identifier starting with *A* will collide with the first starting with *B*, and so on. A better technique is to use bigger steps in looking for the next slot. A fairly effective way is to use steps defined by a moderately small prime number - and, as we have already suggested, to use a symbol table that is itself able to contain a prime number of items. Then in the worst case we shall easily be able to detect that the table is full, while still being able to utilize every available slot before this happens.

The implementation in Appendix D shows how these ideas can be implemented in a table handler compatible with the rest of the assembler. The suggested hashing function is relatively complicated, but is intended to produce a relatively large range of keys. The search itself is programmed using the so-called *state variable* approach: while searching we can be in one of four states - still looking, found the identifier we are looking for, found a free slot, or found that the table is full.

The above discussion may have given the impression that the use of hashing functions is so beset with problems as to be almost useless, but in fact they turn out to be the method of choice for serious applications. If a little care is taken over the choice of hashing function, the collision rate can be kept very low, and the speed of access very high.

Exercises

7.3 How could one make use of a hash table to speed up the process of matching mnemonics to opcodes?

7.4 Could one use a single hash table to store opcode mnemonics, directive mnemonics, macro labels, and user defined labels?

7.5 In the implementation in Appendix D the hash function is computed within the symbol table handler itself. It might be more efficient to compute it as the identifier is recognized within the scanner. What modifications would be needed to the scanner interface to achieve this?

Further reading

Our treatment of hash functions has been very superficial. Excellent treatments of this subject are to be found in the books by Gough (1988), Fischer and LeBlanc (1988, 1991) and Elder (1994).

7.4 Macro processing facilities

Programming in ASSEMBLER is a tedious business at the best of times, because assembler languages are essentially very simple, and lack the powerful constructs possible in high level languages. One way in which life can be made easier for programmers is to permit them to use macros. A **macro** is a device that effectively allows the assembler language to be extended, by the programmer defining new mnemonics that can then be used repeatedly within the program thereafter. As usual, it is useful to have a clear syntactic description of what we have in mind. Consider the following modification to the `PRODUCTIONS` section of the second Cocol grammar of section 6.1, which allows for various of the extensions now being proposed:

```
PRODUCTIONS
  ASM               = StatementSequence "END" EOF .
  StatementSequence = { Statement [ comment ] EOL } .
  Statement         = Executable | MacroExpansion | Directive .
  Executable        = [ Label ] [ OneByteOp | TwoByteOp Address ] .
  OneByteOp         = "HLT" | "PSH" | "POP"  (* | . . . . etc *) .
  TwoByteOp         = "LDA" | "LDX" | "LDI"  (* | . . . . etc *) .
  Address           = Term { "+" Term | "-" Term } .
  Term              = Label | number | "*" .
  MacroExpansion    = [ Label ] MacroLabel ActualParameters .
  ActualParameters  = [ OneActual { "," OneActual } ] .
  OneActual         = Term | OneByteOp | TwoByteOp .
  Directive         =   Label "EQU" KnownAddress
                      | [ Label ] ( "DC" Address | "DS" KnownAddress )
                      | "ORG" KnownAddress | "BEG"
                      | "IF" KnownAddress | MacroDefinition .
  Label             = identifier .
  KnownAddress      = Address .
  MacroDefinition   = MacroLabel "MAC" FormalParameters [ comment ] EOL
                      StatementSequence
                      "END" .
  MacroLabel        = identifier .
  FormalParameters  = [ identifier { "," identifier } ] .
```

Put less formally, we are adopting the convention that a macro is defined by code like

```
 LABEL   MAC     P1, P2, P3 ...  ; P1, P2, P3 ... are formal parameters
                                 ; lines of code as usual,
                                 ; using P1, P2, P3 ... in various fields
         END                     ; end of definition
```

where LABEL is the name of the new instruction, and where MAC is a new directive. For example, we might have

```
SUM     MAC     A,B,C   ; Macro to add A to B and store in C
        LDA     A
        ADD     B
        STA     C
        END             ; of macro SUM
```

It must be emphasized that a macro definition gives a template or model, and does not of itself immediately generate executable code. The program will, in all probability, not have labels or variables with the same names as those given to the formal parameters.

If a program contains one or more macro definitions, we may then use them to generate executable code by a **macro expansion**, which takes a form exemplified by

```
        SUM     X,Y,Z
```

where SUM, the name of the macro, appears in the opcode field, and where X,Y,Z are known as **actual parameters**. With SUM defined as in this example, code of the apparent form

```
        SUM     X,Y,Z
    L1  SUM     P,Q,R
```

would be expanded by the assembly process to generate actual code equivalent to

```
        LDA     X
        ADD     Y
        STA     Z
    L1  LDA     P
        ADD     Q
        STA     R
```

In the example above the formal parameters appeared only in the address fields of the lines constituting the macro definition, but they are not restricted to such use. For example, the macro

```
CHK     MAC     A,B,OPCODE,LAB
LAB     LDA     A
        CPI     B
        OPCODE  LAB
        END             ; of macro CHK
```

if invoked by code of the form

```
        CHK     X,Y,BNZ,L1
```

would produce code equivalent to

```
 L1     LDA     X
        CPI     Y
        BNZ     L1
```

A *macro* facility should not be confused with a *subroutine* facility. The definition of a macro causes no code to be assembled, nor is there any obligation on the programmer ever to expand any particular macro. On the other hand, defining a subroutine *does* cause code to be generated immediately. Whenever a macro is expanded the assembler generates code equivalent to the macro body, but with the actual parameters textually substituted for the formal parameters. For the call of a subroutine the assembler simply generates code for a special form of jump to the subroutine.

We may add a macro facility to a one-pass assembler quite easily, if we stipulate that each macro must be fully defined before it is ever invoked (this is no real limitation if one thinks about it).

The first problem to be solved is that of macro definition. This is easily recognized as imminent by the `assembleline` routine, which handles the `MAC` directive by calling a `definemacro` routine from within the switching construct responsible for handling directives. The `definemacro` routine provides (recursively) for the definition of one macro within the definition of another one, and for fairly sophisticated error handling.

The definition of a macro is handled in two phases. Firstly, an entry must be made into a macro table, recording the name of the macro, the number of parameters, and their formal names. Secondly, provision must be made to store the source text of the macro so that it may be rescanned every time a macro expansion is called for. As usual, in a C++ implementation we can make effective use of yet another class, which we introduce with the following public interface:

```
typedef struct MH_macentries *MH_macro;

class MH {
  public:
    void newmacro(MH_macro &m, SA_unpackedlines header);
    // Creates m as a new macro, with given header line that includes
    // the formal parameters

    void storeline(MH_macro m, SA_unpackedlines line);
    // Adds line to the definition of macro m

    void checkmacro(char *name, MH_macro &m, bool &ismacro, int &params);
    // Checks to see whether name is that of a predefined macro.  Returns
    // ismacro as the result of the search.  If successful, returns m as
    // the macro, and params as the number of formal parameters

    void expand(MH_macro m, SA_addresses actualparams,
                ASMBASE *assembler, bool &errors);
    // Expands macro m by invoking assembler for each line of the macro
    // definition, and using the actualparams supplied in place of the
    // formal parameters appearing in the macro header.
    // errors is altered to true if the assembly fails for any reason

    MH();
    // Initializes macro handler
};
```

The algorithm for assembling an individual line is, essentially, the same as before. The difference is that, before assembly, the `mnemonic` field is checked to see whether it is a user-defined macro name rather than a standard machine opcode. If it is, the macro is expanded, effectively by assembling lines from the text stored in the macro body, rather than from the incoming source.

The implementation of the macro handler class is quite interesting, and calls for some further commentary:

- A variable of `MC_macro` type is simply a pointer to a node from which depends a queue of unpacked source lines. This header node records the unpacked line that forms the macro header itself, and the address field in this header line contains the formal parameters of the macro.

- Macro expansion is accomplished by passing the lines stored in the queue to the same `assembleline` routine that is responsible for assembling "normal" lines. The mutual recursion which this introduces into the system (the assembler has to be able to invoke the macro expansion, which has to be able to invoke the assembler) is handled in a C++ implementation by declaring a small base class

  ```
  class ASMBASE {
    public:
      virtual void assembleline(SA_unpackedlines &srcline, bool &failure) = 0;
      // Assembles srcline, reporting failure if it occurs
  };
  ```

The assembler class is then derived from this one, and the base class is also used as a formal parameter type in the `MH::expand` function. The same sort of functionality is achieved in Pascal and Modula-2 implementations by passing the `assembleline` routine as an actual parameter directly to the `expand` routine.

- The macro expansion has to substitute the actual parameters from the address field of the macro invocation line in the place of any formal parameter references that may appear in each of the lines stored in the macro "body" before those lines can be assembled. These formal parameters may of course appear as labels, mnemonics, or as elements of addresses.

- A macro expansion may instigate another macro expansion - indeed any use of macro processing other than the most trivial probably takes advantage of this feature. Fortunately this is easily handled by the various routines calling one another in a (possibly) mutually recursive manner.

Exercises

7.6 The following represents an attempt to solve a very simple problem:

```
        BEG
CR      EQU     13          ; ASCII carriage return
LF      EQU     10          ; ASCII line feed
WRITE   MAC     A, B, C     ; write integer A and characters B,C
        LDA     A
        OTI                 ; write integer
        LDI     B
        OTA                 ; write character
        LDI     C
        OTA                 ; write character
        END                 ; of WRITE macro
READ    MAC     A
        INI
        STA     A
        WRITE   A, CR, LF ; reflect on output
        END                 ; of READ macro
LARGE   MAC     A, B, C     ; store larger of A,B in C
        LDA     A
        CMP     B
        BPZ     * + 3
        LDA     B
        STA     C
        END                 ; of LARGE macro

        READ    X
        READ    Y
        READ    Z
        LARGE   X, Y, LARGE
        LARGE   LARGE, Z, LARGE
EXIT    WRITE   LARGE, CR, LF
        HLT
LARGE   DS      1
X       DS      1
Y       DS      1
Z       DS      1
        END                 ; of program
```

If this were assembled by our macro assembler, what would the symbol, forward reference and macro tables look like just before the line labelled `EXIT` was assembled? Is it permissible to use the identifier `LARGE` as both the name of a macro and of a label?

7.7 The `LARGE` macro of the last example is a little dangerous, perhaps. Addresses like `* + 3` are apt to cause trouble when modifications are made, because programmers forget to change absolute addresses or offsets. Discuss the implications of coding the body of this macro as

```
        LDA     A
        CMP     B
        BPZ     LAB
        LDA     B
LAB     STA     C
        END             ; of LARGE macro
```

7.8 Develop macros using the language suggested here that will allow you to simulate the **if ... then ... else**, **while ... do**, **repeat ... until**, and **for** loop constructions allowed in high level languages.

7.9 In our system, a macro may be defined *within* another macro. Is there any advantage in allowing this, especially as macros are all entered in a globally accessible macro table? Would it be possible to make nested macros obey scope rules similar to those found in Pascal or Modula-2?

7.10 Suppose two macros use the same formal parameter names. Does this cause problems when attempting macro expansion? Pay particular attention to the problems that might arise in the various ways in which nesting of macro expansions might be required.

7.11 Should one be able to redefine macro names? What does our system do if this is attempted, and how should it be changed to support any ideas you may have for improving it?

7.12 Should the number of formal and actual parameters be allowed to disagree?

7.13 To what extent can a macro assembler be used to accept code in one assembly language and translate it into opcodes for another one?

7.5 Conditional assembly

To realize the full power of an assembler (even one with no macro facilities), it may be desirable to add the facility for what is called **conditional assembly**, whereby the assembler can determine at assembly-time whether to include certain sections of source code, or simply ignore them. A simple form of this is obtained by introducing an extra directive `IF`, used in code of the form

```
        IF      Expression
```

which signals to the assembler that the *following* line is to be assembled only if the *assembly-time* value of *Expression* is non-zero. Frequently this line might be a macro invocation, but it does not have to be. Thus, for example, we might have

```
SUM     MAC     A,B,C
        LDA     A
        ADD     B
        STA     C
        END             ; macro
        . . .
FLAG    EQU     1
        . . .
        IF      FLAG
        SUM     X,Y,RESULT
```

which (in this case) would generate code equivalent to

```
        LDA     X
        ADD     Y
        STA     RESULT
```

but if we had set `FLAG EQU 0` the macro expansion for `SUM` would not have taken place.

This may seem a little silly, and another example may be more convincing. Suppose we have defined the macro

```
SUM   MAC     A,B,C,FLAG
      LDA     A
      IF      FLAG
      ADI     B
      IF      FLAG-1
      ADX     B
      STA     C
      END             ; macro
```

Then if we ask for the expansion

```
      SUM     X,45,RESULT,1
```

we get assembled code equivalent to

```
      LDA     X
      ADI     45
      STA     RESULT
```

but if we ask for the expansion

```
      SUM     X,45,RESULT,0
```

we get assembled code equivalent to

```
      LDA     X
      ADX     45
      STA     RESULT
```

This facility is almost trivially easily added to our one-pass assembler, as can be seen by studying the code for the first few lines of the `AS::assembleline` function in Appendix D (which handles the inclusion or rejection of a line), and the `case AS_if` clause that handles the recognition of the `IF` directive. Note that the value of *Expression* must be completely defined by the time the `IF` directive is encountered, which may be a little more restrictive than we could allow with a two-pass assembler.

Exercises

7.14 Should a macro be allowed to contain a reference to itself? This will allow recursion, in a sense, in assembly language programming, but how does one prevent the system from getting into an indefinite expansion? Can it be done with the facilities so far developed? If not, what must be added to the language to allow the full power of recursive macro calls?

7.15 *N!* can be defined recursively as

> **if** *N* = 1 **then** *N!* = 1 **else** *N! = N(N-1)!*

In the light of your answer to Exercise 7.14, can you make use of this idea to let the macro assembler developed so far generate code for computing 4! by using recursive macro calls?

7.16 Conditional assembly may be enhanced by allowing constructions of the form

```
IF      EXPRESSION
   line 1
   line 2
   . . .
```

```
ENDIF
```

with the implication that the code up to the directive `ENDIF` is only assembled if `EXPRESSION` evaluates to a non-zero result at assembly-time. Is this really a necessary, or a desirable variation? How could it be implemented? Other extensions might allow code like that below (with fairly obvious meaning):

```
IF      EXPRESSION
     line 1
     line 2
     . . .
ELSE
     line m
     line n
     . . .
ENDIF
```

7.17 Conditional assembly might be made easier if one could use Boolean expressions rather than numerical ones. Discuss the implications of allowing, for example

```
IF      A > 0
```

or

```
IF      A <> 0 AND B = 1
```

7.6 Relocatable code

The assemblers that we have considered so far have been load-and-go type assemblers, producing the machine instructions for the absolute locations where they will reside when the code is finally executed. However, when developing a large program it is convenient to be able to assemble it in sections, storing each separately, and finally linking the sections together before execution. To some extent this can be done with our present system, by placing an extra load on programmers to ensure that all the sections of code and data are assembled for different areas in memory, and letting them keep track of where they all start and stop.

This is so trivial that it need be discussed no further here. However, such a scheme, while in keeping with the highly simplified view of actual code generation used in this text, is highly unsatisfactory. More sophisticated systems provide the facility for generating relocatable code, where the decision as to where it will finally reside is delayed until loading time.

At first sight even this seems easy to implement. With each byte that is generated we associate a flag, indicating whether the byte will finally be loaded unchanged, or whether it must be modified at load time by adding an offset to it. For example, the section of code

```
00                BEG
00  19 06         LDA  A
02  22 37         ADI  55
04  1E 07         STA  B
06  0C     A      DC   12
07  00     B      DC   0
08                END
```

contains two bytes (assembled as at 01h and 05h) that refer to addresses which would alter if the code was relocated. The assembler could easily produce output for the loader on the lines of the following (where, as usual, values are given in hexadecimal):

```
19 0    06 1    22 0    37 0    1E 0    07 1    0C 0    00 0
```

Here the first of each pair denotes a loadable byte, and the second is a flag denoting whether the byte needs to be offset at load time. A *relocatable code* file of this sort of information could, again, be preceded by a count of the number of bytes to be loaded. The loader could read a set of such files, effectively concatenating the code into memory from some specified overall starting address, and keeping track as it did so of the offset to be used.

Unfortunately, the full ramifications of this soon reach far beyond the scope of a naïve discussion. The main point of concern is how to decide which bytes must be regarded as relocatable. Those defined by "constants", such as the opcodes themselves, or entries in the symbol table generated by `EQU` directives are clearly "absolute". Entries in the symbol table defined by "labels" in the label field of other instructions may be thought of as relocatable, but bytes defined by expressions that involve the use of such labels are harder to analyse. This may be illustrated by a simple example.

Suppose we had the instruction

```
LDA   A - B
```

If `A` and `B` are absolute, or are both relocatable, and both defined in the section of code being assembled, then the difference is absolute. If `B` is absolute and `A` is relocatable, then the difference is still relocatable. If `A` is absolute and `B` is relocatable, then the difference should probably be ruled inadmissible. Similarly, if we have an instruction like

```
LDA   A + B
```

the sum is absolute if both `A` and `B` are absolute, is relocatable if `A` is relocatable and `B` is absolute, and probably inadmissible otherwise. Similar arguments may be extended to handle an expression with more than two operands (but notice that expressions with multiplication and division become still harder to analyse).

The problem is exacerbated still further if - as will inevitably happen when such facilities are properly exploited - the programmer wishes to make reference to labels which are *not* defined in the code itself, but which may, perhaps, be defined in a separately assembled routine. It is not unreasonable to expect the programmer explicitly to declare the names of all labels to be used in this way, perhaps along the lines of

```
BEG
DEF     A,B,C     ; these are available for external use
USE     X,Y,Z     ; these are not defined, but required
```

In this case it is not hard to see that the information presented to the loader will have to be quite considerably more complex, effectively including those parts of the symbol table relating to the elements of the `DEF` list, and those parts of the forward reference tables that relate to the `USE` list. Rather cowardly, we shall refrain from attempting to discuss these issues in further detail here, but leave them as interesting topics for the more adventurous reader to pursue on his or her own.

7.7 Further projects

The following exercises range from being almost trivial to rather long and involved, but the reader who successfully completes them will have learned a lot about the assembly translation process, and possibly even something about assembly language programming.

7.18 We have discussed extensions to the one-pass assembler, rather than the two-pass assembler.

Attempt to extend the two-pass assembler in the same way.

7.19 What features could you add to, and what restrictions could you remove from the assembly process if you used a two-pass rather than a one-pass assembler? Try to include these extra features in your two-pass assembler.

7.20 Modify your assembler to provide for the generation of relocatable code, and possibly for code that might be handled by a linkage editor, and modify the loader developed in Chapter 4, so as to include a more sophisticated linkage editor.

7.21 How could you prevent programmers from branching to "data", or from treating "instruction" locations as data - assuming that you thought it desirable to do so? (As we have mentioned, assembler languages usually allow the programmer complete freedom in respect of the treatment of identifiers, something which is expressly forbidden in strictly typed languages like Pascal, but which some programmers regard as a most desirable feature of a language.)

7.22 We have carefully chosen our opcode mnemonics for the language so that they are lexically unique. However, some assemblers do not do this. For example, the 6502 assembler as used on the pioneering Apple II microcomputer had instructions like

```
LDA  2      equivalent to our     LDA  2
```

and

```
LDA  #2     equivalent to our     LDI  2
```

that is, an extra character in the address field denoted whether the addressing mode was "direct" or "immediate". In fact it was even more complex than that: the `LDA` mnemonic in 6502 assembler could be converted into one of 8 machine instructions, depending on the exact form of the address field. What differences would it make to the assembly process if you had to cater for such conventions? To make it realistic, study the 6502 assembler mnemonics in detail.

7.23 Another variation on address field notation was provided by the Intel 8080 assembler, which used mnemonics like

```
MOV  A, B         and     MOV  B, A
```

to generate different single byte instructions. How would this affect the assembly process?

7.24 Some assemblers allow the programmer the facility to use "local" labels, which are not really part of a global symbol list. For example, that provided with the UCSD p-System allowed code like

```
LAB    MVI   A, 4
       JMP   $2
       MVI   B, C
$2     NOP
       LHLD  1234
LAB2   XCHG
       POP   H
$2     POP   B
       POP   D
       JMP   $2
LAB3   NOP
```

Here the `$2` label between the `LAB1` and `LAB2` labels and the `$2` label between the `LAB2` and `LAB3` labels are local to those demarcated sections of code. How difficult is it to add this sort of facility to an assembler, and what would be the advantages in having it?

7.25 Develop a one-pass or two-pass macro assembler for the stack-oriented machine discussed in Chapter 4.

7.26 As a more ambitious project, examine the assembler language for a real microprocessor, and write a good macro assembler for it.

8 GRAMMARS AND THEIR CLASSIFICATION

In this chapter we shall explore the underlying ideas behind grammars further, identify some potential problem areas in designing grammars, and examine the ways in which grammars can be classified. Designing a grammar to describe the syntax of a programming language is not merely an interesting academic exercise. The effort is, in practice, usually made so as to be able to aid the development of a translator for the language (and, of course so that programmers who use the language may have a reference to consult when All Else Fails and they have to Read The Instructions). Our study thus serves as a prelude to the next chapter, where we shall address the important problem of parsing rather more systematically than we have done until now.

8.1 Equivalent grammars

As we shall see, not all grammars are suitable as the starting point for developing practical parsing algorithms, and an important part of compiler theory is concerned with the ability to find **equivalent grammars**. Two grammars are said to be equivalent if they describe the same language, that is, can generate exactly the same set of sentences (not necessarily using the same set of sentential forms or parse trees).

In general we may be able to find several equivalent grammars for any language. A distinct problem in this regard is a tendency to introduce far too few non-terminals, or alternatively, far too many. It should not have escaped attention that the names chosen for non-terminals usually convey some semantic implication to the reader, and the way in which productions are written (that is, the way in which the grammar is factorized) often serves to emphasize this still further. Choosing too few non-terminals means that semantic implications are very awkward to discern at all, too many means that one runs the risk of ambiguity, and of hiding the semantic implications in a mass of hard to follow alternatives.

It may be of some interest to give an approximate count of the numbers of non-terminals and productions that have been used in the definition of a few languages:

Language	Non-terminals	Productions
Pascal (Jensen + Wirth report)	110	180
Pascal (ISO standard)	160	300
Edison	45	90
C	75	220
C++	110	270
ADA	180	320
Modula-2 (Wirth)	74	135
Modula-2 (ISO standard)	225	306

8.2 Case study - equivalent grammars for describing expressions

One problem with the grammars found in text books is that, like many complete programs found in text books, their final presentation often hides the thought which has gone into their development. To try to redress the balance, let us look at a typical language construct - arithmetic expressions - and explore several grammars which seem to define them.

Consider the following EBNF descriptions of simple algebraic expressions. One set is left-recursive, while the other is right-recursive:

```
(E1)    Goal        =  Expression .                    (1)
        Expression  =  Term | Term "-" Expression .    (2 ,3)
        Term        =  Factor | Factor "*" Term .      (4, 5)
        Factor      =  "a" | "b" | "c" .               (6, 7, 8)

(E2)    Goal        =  Expression .                    (1)
        Expression  =  Term | Expression "-" Term .    (2, 3)
        Term        =  Factor | Term "*" Factor .      (4, 5)
        Factor      =  "a" | "b" | "c" .               (6, 7, 8)
```

Either of these grammars can be used to derive the string *a - b * c*, and we show the corresponding phrase structure trees in Figure 8.1 below.

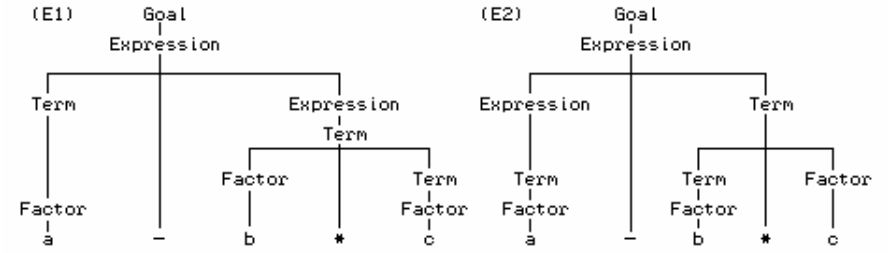


Figure 8.1 Parse trees for the expression a – b * c arising from two grammars

We have already commented that it is frequently the case that the semantic structure of a sentence is reflected in its syntactic structure, and that this is a very useful property for programming language specification. The terminals - and * fairly obviously have the "meaning" of subtraction and multiplication. We can reflect this by drawing the abstract syntax tree (AST) equivalents of the above diagrams; ones constructed essentially by eliding out the names of the non-terminals, as depicted in Figure 8.2. Both grammars lead to the same AST, of course.



Figure 8.2 Abstract syntax trees for the expression a – b * c

The appropriate meaning can then be extracted from such a tree by performing a post-order (LRN) tree walk.

While the two sets of productions lead to the same sentences, the second set of productions corresponds to the usual implied semantics of "left to right" associativity of the operators - and *, while the first set has the awkward implied semantics of "right to left" associativity. We can see this by considering the parse trees for each grammar for the string *a - b - c*, depicted in Figure 8.3.

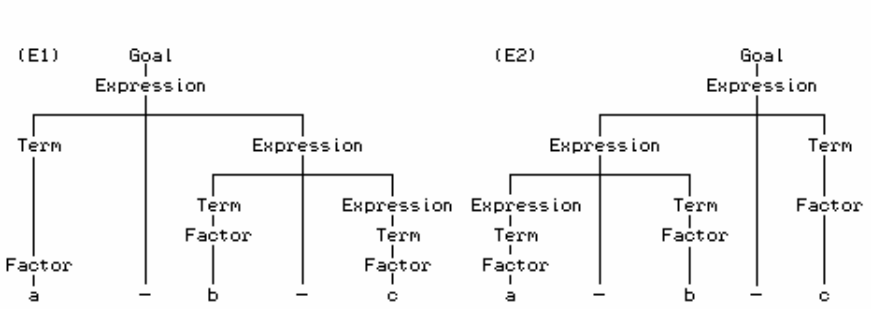


Figure 8.3 Parse trees for the expression a - b - c from two grammars

Another attempt at writing a grammar for this language is of interest:

```
(E3)    Goal        =  Expression .                      (1)
        Expression  =  Term | Term "*" Expression .      (2, 3)
        Term        =  Factor | Factor "-" Term .        (4, 5)
        Factor      =  "a" | "b" | "c" .                 (6, 7, 8)
```

Here we have the unfortunate situation that not only is the associativity of the operators wrong; the relative precedence of multiplication and subtraction has also been inverted from the norm. This can be seen from the parse tree for the expression *a - b * c* shown in Figure 8.4.

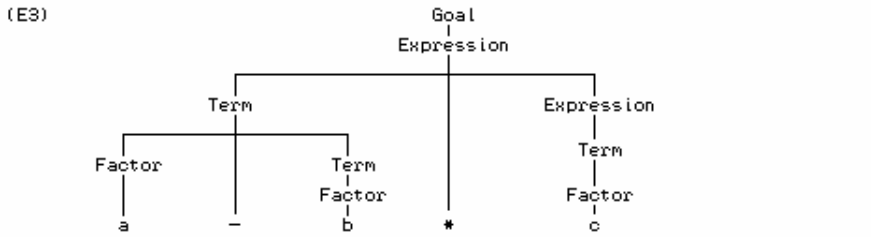


Figure 8.4 Parse tree for the expression a - b * c arising from grammar E3

Of course, if we use the EBNF metasymbols it is possible to write grammars without using recursive productions. Two such grammars follow:

```
(E4)    Goal        =  Expression .                      (1)
        Expression  =  Term { "-" Term } .               (2)
        Term        =  Factor { "*" Factor } .           (3)
        Factor      =  "a" | "b" | "c" .                 (4, 5, 6)

(E5)    Goal        =  Expression .                      (1)
        Expression  =  { Term "-" } Term .               (2)
        Term        =  { Factor  "*" } Factor .          (3)
        Factor      =  "a" | "b" | "c" .                 (4, 5, 6)
```

Exercises

8.1 Draw syntax diagrams which reflect the different approaches taken to factorizing these grammars.

8.2 Comment on the associativity and precedence that seem to underpin grammars E4 and E5.

8.3 Develop sets of productions for algebraic expressions that will describe the operations of addition and division as well as subtraction and multiplication. Analyse your attempts in some detail, paying heed to the issues of associativity and precedence.

8.4 Develop sets of productions which describe expressions exemplified by

$$- a + \sin(b + c) * (- (b - a))$$

that is to say, fairly general mathematical expressions, with bracketing, leading unary signs, the usual operations of addition, subtraction, division and multiplication, and simple function calls. Ensure that the productions correspond to the conventional precedence and associativity rules for arithmetic expressions.

8.5 Extend Exercise 8.4 to allow for exponentiation as well.

8.3 Some simple restrictions on grammars

Had he looked at our grammars, Mr. Orwell might have been tempted to declare that, while they might be equal, some are more equal than others. Even with only limited experience we have seen that some grammars will have features which will make them awkward to use as the basis of compiler development. There are several standard restrictions which are called for by different parsing techniques, among which are some fairly obvious ones.

8.3.1 Useless productions and reduced grammars

For a grammar to be of practical value, especially in the automatic construction of parsers and compilers, it should not contain superfluous rules that cannot be used in parsing a sentence. Detection of useless productions may seem a waste of time, but it may also point to a clerical error (perhaps an omission) in writing the productions. An example of a grammar with useless productions is

$$
\begin{aligned}
&G = \{ N , T , S , P \} \\
&N = \{ W , X , Y , Z \} \\
&T = \{ a \} \\
&S = W \\
&P = \\
&\qquad W \rightarrow aW \qquad (1) \\
&\qquad W \rightarrow Z \qquad (2) \\
&\qquad W \rightarrow X \qquad (3) \\
&\qquad Z \rightarrow aZ \qquad (4) \\
&\qquad X \rightarrow a \qquad (5) \\
&\qquad Y \rightarrow aa \qquad (6)
\end{aligned}
$$

The useful productions are (1), (3) and (5). Production (6) ($Y \rightarrow aa$) is useless, because Y is **non-reachable** or **non-derivable** - there is no way of introducing Y into a sentential form (that is, $S \Rightarrow^{*} \alpha Y \beta$ for any α, β). Productions (2) and (4) are useless, because Z is **non-terminating** - if Z appears in a sentential form then this cannot generate a terminal string (that is, $Z \Rightarrow^{*} \alpha$ for any $\alpha \in T^{*}$).

A **reduced grammar** is one that does not contain superfluous rules of these two types (non-terminals that can never be reached from the start symbol, and non-terminals that cannot produce terminal strings).

More formally, a context-free grammar is said to be reduced if, for each non-terminal B we can write

$$S \Rightarrow^* \alpha B \beta$$

for some strings α and β, and where

$$B \Rightarrow^* \gamma$$

for some $\gamma \in T^*$.

In fact, non-terminals that cannot be reached in any derivation from the start symbol are sometimes added so as to assist in describing the language - an example might be to write, for C

Comment = "/*" *CommentString* "*/" .
CommentString = *character* | *CommentString character* .

8.3.2 ε-free grammars

Intuitively we might expect that detecting the presence of "nothing" would be a little awkward, and for this reason certain compiling techniques require that a grammar should contain no ε-productions (those which generate the null string). Such a grammar is referred to as an ε-free grammar.

ε-productions are usually used in BNF as a way of terminating recursion, and are often easily removed. For example, the productions

```
Integer       =  digit  RestOfInteger .
RestOfInteger =  digit  RestOfInteger | ε .
digit         =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

can be replaced by the ε-free equivalent

```
Integer       =  digit  |  Integer  digit .
digit         =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

Such replacement may not always be so easy: the reader might like to look at the grammar of Section 8.7, which uses ε-productions to express *ConstDeclarations, VarDeclarations* and *Statement*, and try to eliminate them.

8.3.3 Cycle-free grammars

A production in which the right side consists of a single non-terminal

$$A \rightarrow B \quad (\text{ where } A, B \in N)$$

is termed a **single production**. Fairly obviously, a single production of the form

$$A \rightarrow A$$

serves no useful purpose, and should never be present. It could be argued that it causes no harm, for it presumably would be an alternative which was never used (so being useless, in a sense not quite that discussed above). A less obvious example is provided by the set of productions

$$A \rightarrow B$$
$$B \rightarrow C$$

$$C \rightarrow A$$

Not only is this useless in this new sense, it is highly undesirable from the point of obtaining a unique parse, and so all parsing techniques require a grammar to be **cycle-free** - it should not permit a derivation of the form

$$A \Rightarrow^{+} A$$

8.4 Ambiguous grammars

An important property which one looks for in programming languages is that every sentence that can be generated by the language should have a unique parse tree, or, equivalently, a unique left (or right) canonical parse. If a sentence produced by a grammar has two or more parse trees then the grammar is said to be *ambiguous*. An example of ambiguity is provided by another attempt at writing a grammar for simple algebraic expressions - this time apparently simpler than before:

```
(E6)        Goal         =   Expression .                 (1)
            Expression   =   Expression "-" Expression    (2)
                             | Expression "*" Expression  (3)
                             | Factor  .                  (4)
            Factor       =   "a" | "b" | "c" .            (5, 6, 7)
```

With this grammar the sentence *a - b * c* has two distinct parse trees and two canonical derivations. We refer to the numbers to show the derivation steps.

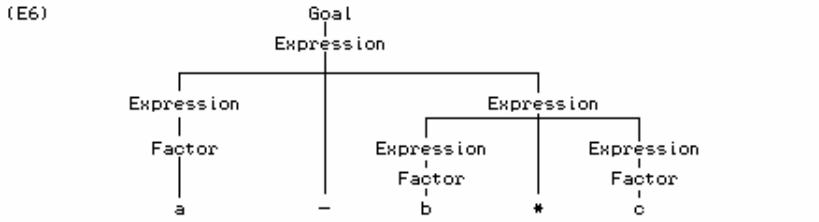


Figure 8.5 One parse tree for the expression a - b * c using grammar E6

The parse tree shown in Figure 8.5 corresponds to the derivation

```
Goal  ⟶  Expression                                  (1)
      ⟶  Expression - Expression                     (2)
      ⟶  Factor - Expression                         (4)
      ⟶  a - Expression                              (5)
      ⟶  a - Expression * Expression                 (3)
      ⟶  a - Factor * Expression                     (4)
      ⟶  a - b * Expression                          (6)
      ⟶  a - b * Factor                              (4)
      ⟶  a - b * c                                   (7)
```

while the second derivation

```
Goal  ⟶  Expression                                  (1)
      ⟶  Expression * Expression                     (3)
      ⟶  Expression - Expression * Expression        (2)
      ⟶  Factor - Expression * Expression            (4)
      ⟶  a - Expression * Expression                 (5)
      ⟶  a - Factor * Expression                     (4)
      ⟶  a - b * Expression                          (6)
      ⟶  a - b * Factor                              (4)
      ⟶  a - b * c                                   (7)
```

corresponds to the parse tree depicted in Figure 8.6.

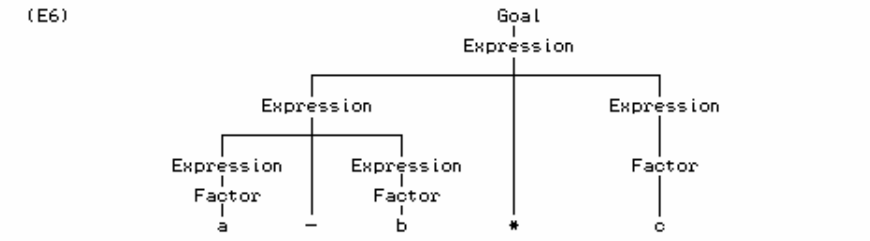


Figure 8.6 Another parse tree for the expression a - b * c using grammar E6

If the only use for grammars was to determine whether a string belonged to the language, ambiguity would be of little consequence. However, if the meaning of a program is to be tied to its syntactic structure, then ambiguity must be avoided. In the example above, the two trees correspond to two different evaluation sequences for the operators `*` and `-` . In the first case the "meaning" would be the usual mathematical one, namely *a* - (*b* * *c*), but in the second case the meaning would effectively be (*a* - *b*) * *c* .

We have already seen various examples of unambiguous grammars for this language in an earlier section, and in this case, fortunately, ambiguity is quite easily avoided.

The most famous example of an ambiguous grammar probably relates to the `IF ... THEN ... ELSE` statement in simple Algol-like languages. Let us demonstrate this by defining a simple grammar for such a construct.

```
Program      =  Statement .
Statement    =  Assignment | IfStatement .
Assignment   =  Variable ":=" Expression .
Expression   =  Variable .
Variable     =  "i" | "j" | "k" | "a" | "b" | "c" .
IfStatement  =  "IF" Condition "THEN" Statement
                | "IF" Condition "THEN" Statement "ELSE" Statement .
Condition    =  Expression "=" Expression
                | Expression "#" Expression .
```

In this grammar the string

```
IF i = j THEN IF i = k THEN a := b ELSE a := c
```

has two possible parse trees. The reader is invited to draw these out as an exercise; the essential point is that we can parse the string to correspond either to

```
IF i = j THEN (IF i = k THEN a := b ELSE a := c)
         ELSE (nothing)
```

or to

```
IF i = j THEN (IF i = k THEN a := b ELSE nothing)
         ELSE (a := c)
```

Any language which allows a sentence such as this may be inherently ambiguous unless certain restrictions are imposed on it, for example, on the part following the `THEN` of an *IfStatement*, as was done in Algol (Naur, 1963). In Pascal and C++, as is hopefully well known, an `ELSE` is deemed to be attached to the most recent unmatched `THEN`, and the problem is avoided that way. In other languages it is avoided by introducing closing tokens like `ENDIF` and `ELSIF`. It is, however, possible to write productions that *are* unambiguous:

```
Statement    =   Matched | Unmatched .
```

```
Matched      =   "IF" Condition "THEN" Matched "ELSE" Matched
                  | OtherStatement .

Unmatched    =   "IF" Condition "THEN" Statement
                  | "IF" Condition "THEN" Matched "ELSE" Unmatched .
```

In the general case, unfortunately, no algorithm exists (or can exist) that can take an arbitrary grammar and determine with certainty and in a finite amount of time whether it is ambiguous or not. All that one can do is to develop fairly simple but non-trivial conditions which, if satisfied by a grammar, assure one that it is unambiguous. Fortunately, ambiguity does not seem to be a problem in practical programming languages.

Exercises

8.6 Convince yourself that the last set of productions for `IF ... THEN ... ELSE` statements is unambiguous.

8.5 Context sensitivity

Some potential ambiguities belong to a class which is usually termed **context-sensitive**. Spoken and written language is full of such examples, which the average person parses with ease, albeit usually within a particular cultural context or idiom. For example, the sentences

Time flies like an arrow

and

Fruit flies like a banana

in one sense have identical construction

Noun Verb Adverbial phrase

but, unless we were preoccupied with aerodynamics, in listening to them we would probably subconsciously parse the second along the lines of

Adjective Noun Verb Noun phrase

Examples like this can be found in programming languages too. In Fortran a statement of the form

A = B(J)

(when taken out of context) could imply a reference either to the Jth element of array B, or to the evaluation of a function B with integer argument J. Mathematically there is little difference - an array can be thought of as a mapping, just as a function can, although programmers may not often think that way.

8.6 The Chomsky hierarchy

Until now all our practical examples of productions have had a single non-terminal on the left side, although grammars may be more general than that. Based on pioneering work by a linguist (Chomsky, 1959), computer scientists now recognize four classes of grammar. The classification depends on the format of the productions, and may be summarized as follows:

8.6.1 Type 0 Grammars (Unrestricted)

An **unrestricted** grammar is one in which there are virtually no restrictions on the form of any of the productions, which have the general form

$$\alpha \rightarrow \beta \qquad \text{with } \alpha \in (N \cup T)^* N (N \cup T)^*, \beta \in (N \cup T)^*$$

(thus the only restriction is that there must be at least one non-terminal symbol on the left side of each production). The other types of grammars are more restricted; to qualify as being of type 0 rather than one of these more restricted types it is necessary for the grammar to contain at least one production $\alpha \rightarrow \beta$ with $|\alpha| > |\beta|$, where $|\alpha|$ denotes the length of α. Such a production can be used to "erase" symbols - for example, *aAB* $\rightarrow$ *aB* erases *A* from the context *aAB*. This type is so rare in computer applications that we shall consider it no further here. Practical grammars need to be far more restricted if we are to base translators on them.

8.6.2 Type 1 Grammars (Context-sensitive)

If we impose the restriction on a type 0 grammar that the number of symbols in the string on the left of any production is less than or equal to the number of symbols on the right side of that production, we get the subset of grammars known as type 1 or **context-sensitive**. In fact, to qualify for being of type 1 rather than of a yet more restricted type, it is necessary for the grammar to contain at least one production with a left side longer than one symbol.

Productions in type 1 grammars are of the general form

$$\alpha \rightarrow \beta \qquad \text{with } |\alpha| \leq |\beta|, \alpha \in (N \cup T)^* N (N \cup T)^*, \beta \in (N \cup T)^+$$

Strictly, it follows that the null string would not be allowed as a right side of any production. However, this is sometimes overlooked, as ε-productions are often needed to terminate recursive definitions. Indeed, the exact definition of "context-sensitive" differs from author to author. In another definition, productions are required to be limited to the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta \qquad \text{with } \alpha, \beta \in (N \cup T)^*, A \in N^+, \gamma \in (N \cup T)^+$$

although examples are often given where productions are of a more general form, namely

$$\alpha A \beta \rightarrow \zeta \gamma \xi \qquad \text{with } \alpha, \beta, \zeta, \xi \in (N \cup T)^*, A \in N^+, \gamma \in (N \cup T)^+$$

(It can be shown that the two definitions are equivalent.) Here we can see the meaning of context-sensitive more clearly - *A* may be replaced by γ when *A* is found in the context of (that is, surrounded by) α and β.

A much quoted simple example of such a grammar is as follows:

```
G = { N , T , S , P }
N = { A , B , C }
T = { a , b , c }
S = A
P =
        A → aABC | abC          (1, 2)
        CB → BC                 (3)
        bB → bb                 (4)
        bC → bc                 (5)
        cC → cc                 (6)
```

Let us derive a sentence using this grammar. *A* is the start string: let us choose to apply production (1)

$A \rightarrow aABC$

and then in this new string choose another production for *A*, namely (2) to derive

$A \rightarrow a\ abC\ BC$

and follow this by the use of (3). (We could also have chosen (5) at this point.)

$A \rightarrow aab\ BC\ C$

We follow this by using (4) to derive

$A \rightarrow aa\ bb\ CC$

followed by the use of (5) to get

$A \rightarrow aab\ bc\ C$

followed finally by the use of (6) to give

$A \rightarrow aabbcc$

However, with this grammar it is possible to derive a sentential form to which no further productions can be applied. For example, after deriving the sentential form

aabCBC

if we were to apply (5) instead of (3) we would obtain

aabcBC

but no further production can be applied to this string. The consequence of such a failure to obtain a terminal string is simply that we must try other possibilities until we find those that yield terminal strings. The consequences for the reverse problem, namely parsing, are that we may have to resort to considerable *backtracking* to decide whether a string is a sentence in the language.

Exercises

8.7 Derive (or show how to parse) the strings

abc and *aaabbbccc*

using the above grammar.

8.8 Show informally that the strings

abbc , *aabc* and *abcc*

cannot be derived using this grammar.

8.9 Derive a context-sensitive grammar for strings of 0's and 1's so that the number of 0's and 1's is the same.

8.10 Attempt to write context-sensitive productions from which the English examples in section 8.5 could be derived.

8.11 An attempt to use context-sensitive productions in an actual computer language was made by Lee (1972), who gave such productions for the `PRINT` statement in BASIC. Such a statement may be described informally as having the keyword `PRINT` followed by an arbitrary number of *Expressions* and *Strings*. Between each pair of *Expressions* a *Separator* is required, but between any other pair (*String* - *Expression*, *String* - *String* or *Expression* - *String*) the *Separator* is optional.

Study Lee's work, criticize it, and attempt to describe the BASIC `PRINT` statement using a context-free grammar.

8.6.3 Type 2 Grammars (Context-free)

A more restricted subset of context-sensitive grammars yields the type 2 or **context-free** grammars. A grammar is context-free if the left side of every production consists of a single non-terminal, and the right side consists of a non-empty sequence of terminals and non-terminals, so that productions have the form

$$\alpha \rightarrow \beta \qquad \text{with } |\alpha| \leq |\beta| , \alpha \in N , \beta \in (N \cup T)^{+}$$

that is

$$A \rightarrow \beta \qquad \text{with } A \in N , \beta \in (N \cup T)^{+}$$

Strictly, as before, no ε-productions should be allowed, but this is often relaxed to allow $\beta \in (N \cup T)^{*}$. Such productions are easily seen to be context-free, because if A occurs in any string, say $\gamma A \delta$, then we may effect a derivation step $\gamma A \delta \Rightarrow \gamma \beta \delta$ without any regard for the particular context (prefix or suffix) in which A occurs.

Most of our earlier examples have been of this form, and we shall consider a larger example shortly, for a complete small programming language.

Exercises

8.12 Develop a context-free grammar that specifies the set of `REAL` decimal literals that may be written in Fortran. Examples of these literals are

```
-21.5      0.25     3.7E-6    .5E7     6E6   100.0E+3
```

8.13 Repeat the last exercise for `REAL` literals in Modula-2 and Pascal, and `float` literals in C++.

8.14 Find a context-free grammar that describes Modula-2 comments (unlike Pascal and C++, these may be nested).

8.15 Develop a context-free grammar that generates all palindromes constructed of the letters *a* and *b* (palindromes are strings that read the same from either end, like *ababbaba*).

8.6.4 Type 3 Grammars (Regular, Right-linear or Left-linear)

Imposing still further constraints on productions leads us to the concept of a type 3 or **regular** grammar. This can take one or other of two forms (but not both at once). It is **right-linear** if the right side of every production consists of zero or one terminal symbols, optionally followed by a single non-terminal, and if the left side is a single non-terminal, so that productions have the form

$A \rightarrow a$ or $A \rightarrow aB$ with $a \in T$, $A, B \in N$

It is **left-linear** if the right side of every production consists of zero or one terminals optionally preceded by a single non-terminal, so that productions have the form

$A \rightarrow a$ or $A \rightarrow Ba$ with $a \in T$, $A, B \in N$

(Strictly, as before, ε productions are ruled out - a restriction often overlooked). A simple example of such a grammar is one for describing binary integers

```
BinaryInteger  = "0" BinaryInteger | "1" BinaryInteger | "0" | "1" .
```

Regular grammars are rather restrictive - local features of programming languages like the definitions of integer numbers and identifiers can be described by them, but not much more. Such grammars have the property that their sentences may be parsed by so-called **finite state automata**, and can be alternatively described by regular expressions, which makes them of theoretical interest from that viewpoint as well.

Exercises

8.16 Can you describe signed integers and Fortran identifiers in terms of regular grammars as well as in terms of context-free grammars?

8.17 Can you develop a regular grammar that specifies the set of `float` decimal literals that may be written in C++?

8.18 Repeat the last exercise for `REAL` literals in Modula-2, Pascal and Fortran.

8.6.5 The relationship between grammar type and language type

It should be clear from the above that type 3 grammars are a subset of type 2 grammars, which themselves form a subset of type 1 grammars, which in turn form a subset of type 0 grammars (see Figure 8.7).

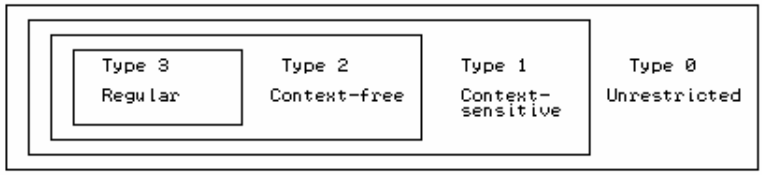


Figure 8.7 The Chomsky hierarchy of grammars

A language $L(G)$ is said to be of type k if it *can* be generated by a type k grammar. Thus, for example, a language is said to be context-free if a context-free grammar may be used to define it. Note that if a non context- free definition is given for a particular language, it does not necessarily imply that the language is not context-free - there may be an alternative (possibly yet-to-be-discovered) context-free grammar that describes it. Similarly, the fact that a language can, for example, most easily be described by a context-free grammar does not necessarily preclude our being able to find an equivalent regular grammar.

As it happens, grammars for modern programming languages are usually largely context-free, with some unavoidable context-sensitive features, which are usually handled with a few extra *ad hoc* rules and by using so- called **attribute grammars**, rather than by engaging on the far more difficult task of finding suitable context- sensitive grammars. Among these features are the following:

- The declaration of a variable must precede its use.
- The number of formal and actual parameters in a procedure call must be the same.
- The number of index expressions or fields in a variable designator must match the number specified in its declaration.

Exercises

8.19 Develop a grammar for describing `scanf` or `printf` statements in C. Can this be done in a context-free way, or do you need to introduce context-sensitivity?

8.20 Develop a grammar for describing Fortran FORMAT statements. Can this be done in a context-free way, or do you need to introduce context-sensitivity?

Further reading

The material in this chapter is very standard, and good treatments of it can be found in many books. The keen reader might do well to look at the alternative presentation in the books by Gough (1988), Watson (1989), Rechenberg and Mössenböck (1989), Watt (1991), Pittman and Peters (1992), Aho,

Sethi and Ullman (1986), or Tremblay and Sorenson (1985). The last three references are considerably more rigorous than the others, drawing several fine points which we have glossed over, but are still quite readable.

8.7 Case study - Clang

As a rather larger example, we give here the complete syntactic specification of a simple programming language, which will be used as the basis for discussion and enlargement at several points in the future. The language is called Clang, an acronym for **C**oncurrent **Lang**uage (also chosen because it has a fine ring to it), deliberately contains a mixture of features drawn from languages like Pascal and C++, and should be immediately comprehensible to programmers familiar with those languages.

The semantics of Clang, and especially the concurrent aspects of the extensions that give it its name, will be discussed in later chapters. It will suffice here to comment that the only data structures (for the moment) are the scalar `INTEGER` and simple arrays of `INTEGER`.

8.7.1 BNF Description of Clang

In the first set of productions we have used recursion to show the repetition:

```
COMPILER Clang

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "(*" TO "*)"

CHARACTERS
  cr         = CHR(13) .
  lf         = CHR(10) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  instring   = ANY - "'" - cr - lf .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  string     = "'" (instring | "''") { instring | "''" } "'" .

PRODUCTIONS
  Clang             = "PROGRAM" identifier ";" Block "." .
  Block             = Declarations CompoundStatement .
  Declarations      = OneDeclaration Declarations | .
  OneDeclaration    = ConstDeclarations | VarDeclarations .
  ConstDeclarations = "CONST" ConstSequence .
  ConstSequence     = OneConst | ConstSequence OneConst .
  OneConst          = identifier "=" number ";" .
  VarDeclarations   = "VAR" VarSequence ";" .
  VarSequence       = OneVar | VarSequence "," OneVar .
  OneVar            = identifier UpperBound .
  UpperBound        = "[" number "]" | .
  CompoundStatement = "BEGIN" StatementSequence "END" .
  StatementSequence = Statement | StatementSequence ";" Statement .
  Statement         =   CompoundStatement | Assignment
                      | IfStatement       | WhileStatement
                      | ReadStatement     | WriteStatement | .
  Assignment        = Variable ":=" Expression .
  Variable          = Designator .
  Designator        = identifier Subscript .
  Subscript         = "[" Expression "]" | .
  IfStatement       = "IF" Condition "THEN" Statement .
  WhileStatement    = "WHILE" Condition "DO" Statement .
  Condition         = Expression RelOp Expression .
  ReadStatement     = "READ" "(" VariableSequence ")" .
  VariableSequence  = Variable | VariableSequence "," Variable .
  WriteStatement    = "WRITE" WriteParameters .
```

```
  WriteParameters   = "(" WriteSequence ")" | .
  WriteSequence     = WriteElement | WriteSequence "," WriteElement .
  WriteElement      = string | Expression .
  Expression        = Term | AddOp Term | Expression AddOp Term .
  Term              = Factor | Term MulOp Factor .
  Factor            = Designator | number | "(" Expression ")" .
  AddOp             = "+" | "-" .
  MulOp             = "*" | "/" .
  RelOp             = "=" | "<>" | "<" | "<=" | ">" | ">=" .
END Clang.
```

8.7.2 EBNF description of Clang

As usual, an EBNF description is somewhat more concise:

```
COMPILER Clang

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "(*" TO "*)"

CHARACTERS
  cr         = CHR(13) .
  lf         = CHR(10) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  instring   = ANY - "'" - cr - lf .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  string     = "'" (instring | "''") { instring | "''" } "'" .

PRODUCTIONS
  Clang             = "PROGRAM" identifier ";" Block "." .
  Block             = { ConstDeclarations | VarDeclarations }
                      CompoundStatement .
  ConstDeclarations = "CONST" OneConst { OneConst } .
  OneConst          = identifier "=" number ";" .
  VarDeclarations   = "VAR" OneVar { "," OneVar } ";" .
  OneVar            = identifier [ UpperBound ] .
  UpperBound        = "[" number "]" .
  CompoundStatement = "BEGIN" Statement { ";" Statement } "END" .
  Statement         = [   CompoundStatement | Assignment
                        | IfStatement       | WhileStatement
                        | ReadStatement     | WriteStatement ] .
  Assignment        = Variable ":=" Expression .
  Variable          = Designator .
  Designator        = identifier [ "[" Expression "]" ] .
  IfStatement       = "IF" Condition "THEN" Statement .
  WhileStatement    = "WHILE" Condition "DO" Statement .
  Condition         = Expression RelOp Expression .
  ReadStatement     = "READ" "(" Variable { "," Variable } ")" .
  WriteStatement    = "WRITE"
                      [ "(" WriteElement { "," WriteElement }  ")" ] .
  WriteElement      = string | Expression .
  Expression        = ( "+" Term | "-" Term | Term ) { AddOp Term } .
  Term              = Factor { MulOp Factor } .
  Factor            = Designator | number | "(" Expression ")" .
  AddOp             = "+" | "-" .
  MulOp             = "*" | "/" .
  RelOp             = "=" | "<>" | "<" | "<=" | ">" | ">=" .
END Clang.
```

8.7.3 A sample program

It is fairly common practice to illustrate a programming language description with an example of a program illustrating many of the language's features. To keep up with tradition, we follow suit. The rather obtuse way in which `Eligible` is incremented before being used in a subscripting expression in line 16 is simply to illustrate that a subscript can be an expression.

```
 PROGRAM Debug;
   CONST
     VotingAge = 18;
   VAR
     Eligible, Voters[100], Age, Total;
```

```
BEGIN
  Total := 0;
  Eligible := 0;
  READ(Age);
  WHILE Age > 0 DO
    BEGIN
      IF Age > VotingAge THEN
        BEGIN
          Voters[Eligible] := Age;
          Eligible := Eligible + 1;
          Total := Total + Voters[Eligible - 1]
        END;
      READ(Age);
    END;
  WRITE(Eligible, ' voters.  Average age = ', Total / Eligible);
END.
```

Exercises

8.21 Do the BNF style productions use right or left recursion? Write an equivalent grammar which uses the opposite form of recursion.

8.22 Develop a set of syntax diagrams for Clang (see section 5.10).

8.23 We have made no attempt to describe the semantics of programs written in Clang; to a reader familiar with similar languages they should be self-evident. Write simple programs in the language to:

(a) Find the sum of the numbers between two input data, which can be supplied in either order.

(b) Use Euclid's algorithm to find the HCF of two integers.

(c) Determine which of a set of year dates correspond to leap years.

(d) Read a sequence of numbers and print out the embedded monotonic increasing sequence.

(e) Use a "sieve" algorithm to determine which of the numbers less than 255 are prime.

In the light of your experience in preparing these solutions, and from the intuition which you have from your background in other languages, can you foresee any gross deficiencies in Clang as a language for handling problems in integer arithmetic (apart from its lack of procedural facilities, which we shall deal with in a later chapter)?

8.24 Suppose someone came to you with the following draft program, seeking answer to the questions currently found in the comments next to some statements. How many of these questions can you answer by referring *only* to the syntactic description given earlier? (The program is not supposed to do anything useful!)

```
PROGRAM Query;
  CONST
    Header = 'Title';  (* Can I declare a string constant? *)
  VAR
    L1[10], L2[10],    (* Are these the same size? *)
    L3[20], I, Query,  (* Can I reuse the program name as a variable? *)
    L3[15];            (* What happens if I use a variable name again? *)
  CONST                (* Can I declare constants after variables? *)
    Max = 1000;
```

```
  Min = -89;         (* Can I define negative constants? *)
VAR                  (* Can I have another variable section? *)
  BigList[Max];      (* Can I use named constants to set array sizes? *)
BEGIN
  Write(Heading)     (* Can I write constants? *)
  L1[10] := 34;      (* Does L[10] exist? *)
  L1 := L2;          (* Can I copy complete arrays? *)
  Write(L3);         (* Can I write complete arrays? *)
  ;; I := Query;;;   (* What about spurious semicolons? *)
END.
```

8.25 As a more challenging exercise, consider a variation on Clang, one that resembles C++ rather more closely than it does Pascal. Using the translation below of the sample program given earlier as a guide, derive a grammar that you think describes this language (which we shall later call "Topsy"). For simplicity, regard `cin` and `cout` as keywords leading to special statement forms.

```
void main (void) {
  const VotingAge = 18;
  int Eligible, Voters[100], Age, Total;

  Total = 0;
  Eligible = 0;
  cin >> Age;
  while (Age > 0) {
    if (Age > VotingAge) {
      Voters[Eligible] = Age;
      Eligible = Eligible + 1;
      Total = Total + Voters[Eligible - 1];
    }
    cin >> Age;
  }
  cout << Eligible << " voters.  Average age = " << Total / Eligible;
}
```

8.26 In the light of your experience with Exercises 8.24 and 8.25, discuss the ease of "reverse-engineering" a programming language description by consulting only a few example programs? Why do you suppose so many students attempt to learn programming by imitation?

8.27 Modify the Clang language definition to incorporate Pascal-like forms of:

(a) the `REPEAT ... UNTIL` statement
(b) the `IF ... THEN ... ELSE` statement
(c) the `CASE` statement
(d) the `FOR` loop
(e) the `MOD` operator.

8.28 Repeat the last exercise for the language suggested by Exercise 8.25, using syntax that resembles that found in C++.

8.29 In Modula-2, structured statements are each terminated with their own `END`. How would you have to change the Clang language definition to use Modula-2 forms for the existing statements, and for the extensions suggested in Exercise 8.27? What advantages, if any, do these forms have over those found in Pascal or C++?

8.30 Study how the specification of string tokens has been achieved in Cocol. Some languages, like Modula- 2, allow strings to be delimited by either single or double quotes, but not to contain the delimiter as a member of the string (so that we might write "David’s Helen’s brother" or ’He said "Hello"’, but not ’He said "That’s rubbish!"’). How would you specify string tokens if these had to match those found in Modula-2, or those found in C++ (where various escape characters are allowed within the string)?

9 DETERMINISTIC TOP-DOWN PARSING

In this chapter we build on the ideas developed in the last one, and discuss the relationship between the formal definition of the syntax of a programming language, and the methods that can be used to parse programs written in that language. As with so much else in this text, our treatment is introductory, but detailed enough to make the reader aware of certain crucial issues.

9.1 Deterministic top-down parsing

The task of the front end of a translator is, of course, not the generation of sentences in a source language, but the recognition of them. This implies that the generating steps which led to the construction of a sentence must be deduced from the finished sentence. How difficult this is to do depends on the complexity of the production rules of the grammar. For Pascal-like languages it is, in fact, not too bad, but in the case of languages like Fortran and C++ it becomes quite complicated, for reasons that may not at first be apparent.

Many different methods for parsing sentences have been developed. We shall concentrate on a rather simple, and yet quite effective one, known as **top-down parsing** by **recursive descent**, which can be applied to Pascal, Modula-2, and many similar languages, including the simple one of section 8.7.

The reason for the phrase "by recursive descent" will become apparent later. For the moment we note that top- down methods effectively start from the goal symbol and try to regenerate the sentence by applying a sequence of appropriate productions. In doing this they are guided by looking at the next terminal in the string that they have been given to parse.

To illustrate top-down parsing, consider the toy grammar

```
G = { N , T , S , P }
N = { A , B }
T = { x , y , z }
S = A
P =
    A  →  xB          (1)
    B  →  z           (2)
    B  →  yB          (3)
```

Let us try to parse the sentence *xyyz*, which clearly is formed from the terminals of this grammar. We start with the goal symbol and the input string

```
Sentential form  S = A            Input string      xyyz
```

To the sentential form *A* we apply the only possible production (1) to get

```
Sentential form     xB            Input string     xyyz
```

So far we are obviously doing well. The leading terminals in both the sentential form and the input string match, and we can effectively discard them from both; what then remains implies that from the non-terminal *B* we must be able to derive *yyz*.

```
Sentential form     B             Input string     yyz
```

We could choose either of productions (2) or (3) in handling the non-terminal *B*; simply looking at the input string indicates that (3) is the obvious choice. If we apply this production we get

```
Sentential form     yB            Input string     yyz
```

which implies that from the non-terminal *B* we must be able to derive *yz*.

```
Sentential form     B             Input string     yz
```

Again we are led to use production (3) and we get

```
Sentential form     yB            Input string     yz
```

which implies that from the non-terminal *B* we must be able to derive the terminal *z* directly - which of course we can do by applying (2).

The reader can easily verify that a sentence composed only of the terminal *x* (such as *xxxx*) could not be derived from the goal symbol, nor could one with *y* as the rightmost symbol, such as *xyyyy*.

The method we are using is a special case of so-called **LL(*k*) parsing**. The terminology comes from the notion that we are scanning the input string from **L**eft to right (the first L), applying productions to the **L**eftmost non- terminal in the sentential form we are manipulating (the second L), and looking only as far ahead as the next *k* terminals in the input string to help decide which production to apply at any stage. In our example, fairly obviously, $k = 1$; LL(1) parsing is the most common form of LL(*k*) parsing in practice.

Parsing in this way is not always as easy, as is evident from the following example

```
G = { N , T , S , P }
N = { A , B , C }
T = { x , y , z }
S = A
P =
    A  →  xB          (1)
    A  →  xC          (2)
    B  →  xB          (3)
    B  →  y           (4)
    C  →  xC          (5)
    C  →  z           (6)
```

If we try to parse the sentence *xxxz* we might proceed as follows

```
Sentential form     S  =  A       Input string     xxxz
```

In manipulating the sentential form *A* we must make a choice between productions (1) and (2). We do not get any real help from looking at the first terminal in the input string, so let us try production (1). This leads to

```
Sentential form     xB            Input string     xxxz
```

which implies that we must be able to derive *xxz* from *B*. We now have a much clearer choice; of the productions for *B* it is (3) which will yield an initial *x*, so we apply it and get to

```
Sentential form     xB            Input string     xxz
```

which implies that we must be able to derive *xz* from *B*. If we apply (1) again we get

```
Sentential form     xB            Input string     xz
```

which implies that we must be able to derive *z* directly from *B*, which we cannot do. If we reflect on this we see that either we cannot derive the string, or we made a wrong decision somewhere along

the line. In this case, fairly obviously, we went wrong right at the beginning. Had we used production (2) and not (1) we should have matched the string quite easily.

When faced with this sort of dilemma, a parser might adopt the strategy of simply proceeding according to one of the possible options, being prepared to retreat along the chosen path if no further progress is possible. Any **backtracking** action is clearly inefficient, and even with a grammar as simple as this there is almost no limit to the amount of backtracking one might have to be prepared to do. One approach to language design suggests that syntactic structures which can only be described by productions that run the risk of requiring backtracking algorithms should be identified, and avoided.

This may not be possible after the event of defining a language, of course - Fortran is full of examples where it seems backtracking might be needed. A classic example is found in the pair of statements

```
DO 10 I = 1 , 2
```

and

```
DO 10 I = 1 . 2
```

These are distinguishable as examples of two totally different statement types (`DO` statement and `REAL` assignment) only by the period/comma. This kind of problem is avoided in modern languages by the introduction of reserved keywords, and by an insistence that white space appear between some tokens (neither of which are features of Fortran, but neither of which cause difficulties for programmers who have never known otherwise).

The consequences of backtracking for full-blooded translators are far more severe than our simple example might suggest. Typically these do not simply read single characters (even "unreading" characters is awkward enough for a computer), but also construct explicit or implicit trees, generate code, create symbol tables and so on - all of which may have to be undone, perhaps just to be redone in a very slightly different way. In addition, backtracking makes the detection of malformed sentences more complicated. All in all, it is best avoided.

In other words, we should like to be able to confine ourselves to the use of *deterministic* parsing methods, that is, ones where at each stage we can be sure of which production to apply next - or, where, if we cannot find a production to use, we can be sure that the input string is malformed.

It might occur to the reader that some of these problems - including some real ones too, like the Fortran example just given - could be resolved by looking ahead more than one symbol in the input string. Perhaps in our toy problem we should have been prepared to scan four symbols ahead? A little more reflection shows that even this is quite futile. The language which this grammar generates can be described by:

$$L(G) = \{ x^n p \mid n > 0, p \in \{y , z\} \}$$

or, if the reader prefers less formality:

> "at least one, but otherwise as many *x*'s in a row as you like, followed by a single *y* or *z*"

We note that being prepared to look more than one terminal ahead is a strategy which can work well in some situations (Parr and Quong, 1996), although, like backtracking, it will clearly be more difficult to implement.

9.2 Restrictions on grammars so as to allow LL(1) parsing

The top-down approach to parsing looks so promising that we should consider what restrictions have to be placed on a grammar so as to allow us to use the LL(1) approach (and its close cousin, the method of recursive descent). Once these have been established we shall pause to consider the effects they might have on the design or specification of "real" languages.

A little reflection on the examples above will show that the problems arise when we have alternative productions for the next (left-most) non-terminal in a sentential form, and should lead to the insight that the *initial* symbols that can be derived from the alternative right sides of the production for a given non-terminal must be distinct.

9.2.1 Terminal start sets, the FIRST function and LL(1) conditions for ε-free grammars

To enhance the discussion, we introduce the concept of the **terminal start symbols** of a non-terminal: the set FIRST(A) of the non-terminal A is defined to be the set of all terminals with which a string derived from A can start, that is

$$a \in \text{FIRST}(A) \qquad \text{if } A \Rightarrow^{+} a\zeta \quad 9\,(A \in N\,;\, a \in T\,;\, \zeta \in (N \cup T)^{*})$$

ε-productions, as we shall see, are a source of complication; for the moment we note that for a unique production of the form $A \rightarrow \varepsilon$, FIRST($A$) = Ø.

In fact we need to go further, and so we introduce the related concept of the terminal start symbols of a general string ξ in a similar way, as the set of all terminals with which ξ or a string derived from ξ can start, that is

$$a \in \text{FIRST}(\xi) \qquad \text{if } \xi \Rightarrow^{*} a\zeta \qquad (a \in T\,;\, \xi, \zeta \in (N \cup T)^{*})$$

again with the *ad hoc* rule that FIRST(ε) = Ø. Note that ε is not a member of the terminal vocabulary T, and that it is important to distinguish between FIRST(ξ) and FIRST(A). The string ξ might consist of a single non- terminal A, but in general it might be a concatenation of several symbols.

With the aid of these we may express a rule that easily allows us to determine when an ε-free grammar is LL(1):

Rule 1

When the productions for any non-terminal A admit alternatives

$$A \rightarrow \xi_1 \mid \xi_2 \mid \ldots \xi_n$$

but where $\xi_k \not\Rightarrow \varepsilon$ for any k, the sets of initial terminal symbols of all strings that can be generated from each of the 's must be disjoint, that is

generated from each of the ξ_k's must be disjoint, that is

$$\text{FIRST}(\xi_j) \cap \text{FIRST}(\xi_k) = \emptyset \quad \text{for all } j \neq k$$

If all the alternatives for a non-terminal A were simply of the form

$$\xi_k = a_k \zeta_k \qquad (a_k \in T ; \xi_k , \zeta_k \in (N \cup T)^*)$$

it would be easy to check the grammar very quickly. All productions would have right-hand sides starting with a terminal, and obviously $\text{FIRST}(a_k\zeta_k) = \{ a_k \}$.

It is a little restrictive to expect that we can write or rewrite all productions with alternatives in this form. More likely we shall find several alternatives of the form

$$\xi_k = B_k\zeta_k$$

where B_k is another non-terminal. In this case to find $\text{FIRST}(B_k\zeta_k)$ we shall have to consider the production rules for B_k, and look at the first terminals which can arise from those (and so it goes on, because there may be alternatives all down the line). All of these must be added to the set $\text{FIRST}(\xi_k)$. Yet another complication arises if B_k is **nullable**, that is, if $B_k \Rightarrow^* \varepsilon$, because in that case we have to add $\text{FIRST}(\zeta_k)$ into the set $\text{FIRST}(\xi_k)$ as well.

The whole process of finding the required sets may be summarized as follows:

- If the first symbol of the right-hand string ξ_k is a terminal, then $\text{FIRST}(\xi_k)$ is of the form $\text{FIRST}(a_k\zeta_k)$, and then $\text{FIRST}(a_k\zeta_k) = \{ a_k \}$.

- If the first symbol of the right-hand string ξ_k is a non-terminal, then $\text{FIRST}(\xi_k)$ is of the form $\text{FIRST}(B_k\zeta_k)$. If B_k is a non-terminal with the derivation rule

  $$B_k \rightarrow \alpha_{k1} \mid \alpha_{k2} \mid \ldots . \mid \alpha_{kn}$$

  then

  $$\text{FIRST}(\xi_k) = \text{FIRST}(B_k\zeta_k) = \text{FIRST}(\alpha_{k1}) \cup \text{FIRST}(\alpha_{k2}) \ldots \cup \text{FIRST}(\alpha_{kn})$$

  with the addition that if any α_{kj} is capable of generating the null string, then the set $\text{FIRST}(\zeta_k)$ has to be included in the set $\text{FIRST}(\xi_k)$ as well.

We can demonstrate this with another toy grammar, rather similar to the one of the last section. Suppose we have

```
G = { N , T , S , P }
N = { A , B , C }
T = { x , y , z }
```

```
S = A
P =
    A  →  B          (1)
    A  →  C          (2)
    B  →  xB         (3)
    B  →  y          (4)
    C  →  xC         (5)
    C  →  z          (6)
```

This generates exciting sentences with any number of *x*'s, followed by a single *y* or *z*. On looking at the alternatives for the non-terminal *A* we see that

$$\text{FIRST}(A_1) = \text{FIRST}(B) = \text{FIRST}(xB) \cup \text{FIRST}(y) = \{ x , y \}$$
$$\text{FIRST}(A_2) = \text{FIRST}(C) = \text{FIRST}(xC) \cup \text{FIRST}(z) = \{ x , z \}$$

so that Rule 1 is violated, as both FIRST(*B*) and FIRST(*C*) have *x* as a member.

9.2.2 Terminal successors, the FOLLOW function, and LL(1) conditions for non ε-free grammars

We have already commented that ε-productions might cause difficulties in parsing. Indeed, Rule 1 is not strong enough to detect another source of trouble, which may arise if such productions are used. Consider the grammar

```
G = { N , T , S , P }
N = { A , B }
T = { x , y }
S = A
P =
    A  →  Bx     (1)
    B  →  xy     (2)
    B  →  ε      (3)
```

In terms of the discussion above, Rule 1 is satisfied. Of the alternatives for the non-terminal *B*, we see that

$$\text{FIRST}(B_1) = \text{FIRST}(xy) = x$$
$$\text{FIRST}(B_2) = \text{FIRST}(\varepsilon) = \emptyset$$

which are disjoint. However, if we try to parse the string *x* we may come unstuck

```
Sentential form   S = A          Input string   x
Sentential form   Bx             Input string   x
```

As we are working from left to right and have a non-terminal on the left we substitute for *B*, to get, perhaps

```
Sentential form   xyx             Input string   x
```

which is clearly wrong. We should have used (3), not (2), but we had no way of telling this on the basis of looking at only the next terminal in the input.

This situation is called the **null string problem**, and it arises only for productions which can generate the null string. One might try to rewrite the grammar so as to avoid ε-productions, but in fact that is not always necessary, and, as we have commented, it is sometimes highly inconvenient. With a little insight we should be able to see that if a non-terminal is nullable, we need to examine the terminals that might legitimately follow it, before deciding that the ε-production is to be applied. With this in mind it is convenient to define the **terminal successors** of a non-terminal *A* as the set of all terminals that can follow *A* in any sentential form, that is

$a \in$ FOLLOW(A) if $S \Rightarrow^* \xi A a \zeta$ ($A, S \in N$; $a \in T$; $\xi, \zeta \in (N \cup T)^*$)

To handle this situation, we impose the further restriction

Rule 2

When the productions for a non-terminal A admit alternatives

$$A \rightarrow \xi_1 \mid \xi_2 \mid \ldots \xi_n$$

and in particular where $\xi_k \Rightarrow \varepsilon$ for some k, the sets of initial terminal symbols of all sentences that can be generated from each of the ξ_j for $j \neq k$ must be disjoint from the set FOLLOW(A) of symbols that may follow any sequence generated from A, that is

$$\text{FIRST}(\xi_j) \cap \text{FOLLOW}(A) = \emptyset, \qquad j \neq k$$

or, rather more loosely,

$$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$$

where, as might be expected

$$\text{FIRST}(A) = \text{FIRST}(\xi_1) \cup \text{FIRST}(\xi_2) \cup \ldots \text{FIRST}(\xi_n)$$

In practical terms, the set FOLLOW(A) is computed by considering every production P_k of the form

$$P_k \rightarrow \xi_k A \zeta_k$$

and forming the sets FIRST(ζ_k), when

$$\text{FOLLOW}(A) = \text{FIRST}(\zeta_1) \cup \text{FIRST}(\zeta_2) \cup \ldots \cup \text{FIRST}(\zeta_n)$$

with the addition that if any ζ_k is also capable of generating the null string, then the set FOLLOW(P_k) has to be included in the set FOLLOW(A) as well.

In the example given earlier, Rule 2 is clearly violated, because

$$\text{FIRST}(B_1) = \text{FIRST}(xy) = \{ x \} = \text{FOLLOW}(B)$$

9.2.3 Further observations

It is important to note two points that may have slipped the reader's attention:

- In the case where the grammar allows ε-productions as alternatives, Rule 2 applies *in addition*

to Rule 1. Although we stated Rule 1 as applicable to ε-free grammars, it is in fact a necessary (but not sufficient) condition that *any* grammar must meet in order to satisfy the LL(1) conditions.

- FIRST is a function that may be applied to a string (in general) and to a non-terminal (in particular), while FOLLOW is a function that is applied to a non-terminal (only).

It may be worth studying a further example so as to explore these rules further. Consider the language defined by the grammar

$$
\begin{aligned}
&G = \{ N , T , S , P \} \\
&N = \{ A , B , C , D \} \\
&T = \{ w , x , y , z \} \\
&S = A \\
&P = \\
&\quad A \rightarrow BD \mid CB && (1, 2) \\
&\quad B \rightarrow x\,B\,z \mid y \mid \varepsilon && (3, 4, 5) \\
&\quad C \rightarrow w \mid z && (6, 7) \\
&\quad D \rightarrow x \mid z && (8, 9)
\end{aligned}
$$

All four non-terminals admit to alternatives, and *B* is capable of generating the empty string ε. Rule 1 is clearly satisfied for the alternative productions for *B*, *C* and *D*, since these alternatives all produce sentential forms that start with distinctive terminals.

To check Rule 1 for the alternatives for *A* requires a little more work. We need to examine the intersection of FIRST(*BD*) and FIRST(*CB*).

FIRST(*CB*) is simply FIRST(*C*) = { *w* } ∪ { *z* } = { *w* , *z* }.

FIRST(*BD*) is not simply FIRST(*B*), since *B* is nullable. Applying our rules to this situation leads to the result that FIRST(*BD*) = FIRST(*B*) ∪ FIRST(*D*) = ({ *x* } ∪ { *y* }) ∪ ({ *x* } ∪ { *z* }) = { *x* , *y* , *z* }.

Since FIRST(*CB*) ∩ FIRST(*BD*) = { *z* }, Rule 1 is broken and the grammar is non-LL(1). Just for completeness, let us check Rule 2 for the productions for *B*. We have already noted that FIRST(*B*) = { *x* , *y* }. To compute FOLLOW(*B*) we need to consider all productions where *B* appears on the right side. These are productions (1), (2) and (3). This leads to the result that

$$
\begin{aligned}
\text{FOLLOW}(B) &= \text{FIRST}(D) && \text{(from the rule } A \rightarrow BD\text{)} \\
&\quad \cup\ \text{FOLLOW}(A) && \text{(from the rule } A \rightarrow CB\text{)} \\
&\quad \cup\ \text{FIRST}(z) && \text{(from the rule } B \rightarrow xBz\text{)} \\
&= \{ x , z \} \cup \emptyset \cup \{ z \} = \{ x , z \}
\end{aligned}
$$

Since FIRST(*B*) ∩ FOLLOW(*B*) = { *x* , *y* } ∩ { *x* , *z* } = { *x* }, Rule 2 is broken as well.

The rules derived in this section have been expressed in terms of regular BNF notation, and we have so far avoided discussing whether they might need modification in cases where the productions are expressed in terms of the option and repetition (closure) metasymbols ([ ] and { } respectively). While it is possible to extend the discussion further, it is not really necessary, in a theoretical sense, to do so. Grammars that are expressed in terms of these symbols are easily rewritten into standard BNF by the introduction of extra non-terminals. For example, the set of productions

$$A \rightarrow \alpha [\xi] \gamma$$

$$B \rightarrow \sigma \{ \zeta \} \tau$$

is readily seen to be equivalent to

$A \rightarrow \alpha\ C\ \gamma$
$B \rightarrow \sigma\ D\ \tau$
$C \rightarrow \xi \mid \varepsilon$
$D \rightarrow \zeta\ D \mid \varepsilon$

to which the rules as given earlier are easily applied (note that the production for *D* is right recursive). In effect, of course, these rules amount to saying for this example that

$\text{FIRST}(\xi) \cap \text{FIRST}(\gamma) = \varnothing$
$\text{FIRST}(\zeta) \cap \text{FIRST}(\tau) = \varnothing$

with the proviso that if γ or τ are nullable, then we must add conditions like

$\text{FIRST}(\xi) \cap \text{FOLLOW}(A) = \varnothing$
$\text{FIRST}(\zeta) \cap \text{FOLLOW}(B) = \varnothing$

There are a few other points that are worth making before closing this discussion.

The reader can probably foresee that in a really large grammar one might have to make many iterations over the productions in forming all the FIRST and FOLLOW sets and in checking the applications of all these rules. Fortunately software tools are available to help in this regard - any reasonable LL(1) compiler generator like Coco/R must incorporate such facilities.

A difficulty might come about in automatically applying the rules to a grammar with which it is possible to derive the empty string. A trivial example of this is provided by

```
G = { N , T , S , P }
N = { A }
T = { x , y }
S = A
P =
    A  →  xy     (1)
    A  →  ε      (2)
```

Here the nullable non-terminal *A* admits to alternatives. In trying to determine FOLLOW(*A*) we should reach the uncomfortable conclusion that this was not really defined, as there are no productions in which *A* appears on the right side. Situations like this are usually handled by constructing a so-called **augmented grammar**, by adding a new terminal symbol (denoted, say, by #), a new goal symbol, and a new single production. For the above example we would create an augmented grammar on the lines of

```
G = { N , T , S , P }
N = { A, B }
T = { x , y, # }
S = B
P =
    B  →  A #    (1)
    A  →  xy     (2)
    A  →  ε      (3)
```

The new terminal # amounts to an explicit end-of-file or end-of-string symbol; we note that realistic parsers and scanners must always be able to detect and react to an end-of-file in a sensible way, so that augmenting a grammar in this way really carries no practical overheads.

9.2.4 Alternative formulations of the LL(1) conditions

The two rules for determining whether a grammar is LL(1) are sometimes found stated in other ways (which are, of course, equivalent). Some authors combine them as follows:

Combined LL(1) Rule

A grammar is LL(1) if for every non-terminal *A* that admits alternatives

$$A \rightarrow \xi_1 \mid \xi_2 \mid \ldots \xi_n$$

the following holds

$$\text{FIRST}(\xi_j \circ \text{FOLLOW}(A)) \cap \text{FIRST}(\xi_k \circ \text{FOLLOW}(A)) = \emptyset, \quad j \neq k$$

where ° denotes "composition" in the mathematical sense. Here the cases $\alpha_j \not\Rightarrow \varepsilon$ and $\alpha_j \Rightarrow \varepsilon$ are combined - for $\alpha_j \not\Rightarrow^* \varepsilon$ we have that $\text{FIRST}(\xi_j \circ \text{FOLLOW}(A)) = \text{FIRST}(\xi_j)$, while for $\alpha_j \Rightarrow^* \varepsilon$ we have similarly that $\text{FIRST}(\xi_j \circ \text{FOLLOW}(A)) = \text{FOLLOW}(A)$.

Other authors conduct this discussion in terms of the concept of **director sets**. For every non-terminal *A* that admits to alternative productions of the form

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

we define $\text{DS}(A, \alpha_k)$ for each alternative to be the set which helps choose whether to use the alternative; when the input string contains the terminal *a* we choose α_k such that $a \in \text{DS}(A, \alpha_k)$. The LL(1) condition is then

$$\text{DS}(A, \alpha_j) \cap \text{DS}(A, \alpha_k) = \emptyset, \quad j \neq k$$

The director sets are found from the relation

$a \in \text{DS}(A, \alpha_k)$ if either $a \in \text{FIRST}(\alpha_k)$ (if $\alpha_k \not\Rightarrow^* \varepsilon$)
or $a \in \text{FOLLOW}(A)$ (if $\alpha_k \Rightarrow^* \varepsilon$)

Exercises

9.1 Test the following grammar for being LL(1)

```
G = { N , T , S , P }
N = { A , B }
T = { w , x , y , z }
S = A
P =
    A  →  B ( x | z ) | ( w | z ) B
    B  →  x B z  | { y }
```

9.2 Show that the grammar describing EBNF itself (section 5.9.1) is LL(1).

9.3 The grammar for EBNF as presented in section 5.9.1 does not allow an implicit ε to appear in a production, although the discussion in that section implied that this was often found in practice. What change could you make to the grammar to allow an implicit ε? Is your resulting grammar still LL(1)? If not, can you find a formulation that *is* LL(1)?

9.4 In section 8.7.2, constant declarations in Clang were described by the productions

```
ConstDeclarations = "CONST" OneConst { OneConst } .
OneConst          = identifier  "="  number  ";" .
```

Is this part of the grammar LL(1)? What would be the effect if one were to factorize the grammar

```
ConstDeclarations = "CONST" OneConst { ";" OneConst } ";" .
OneConst          = identifier  "="  number  .
```

9.5 As a more interesting example of applying an analysis to a grammar expressed in EBNF, let us consider how we might describe the theatrical production of a Shakespearian play with five acts. In each act there may be several scenes, and in each scene appear one or more actors, who gesticulate and make speeches to one another (for the benefit of the audience, of course). Actors come onto the stage at the start of each scene, and come and go as the scene proceeds - to all intents and purposes between speeches - finally leaving at the end of the scene (in the Tragedies some may leave dead, but even these usually revive themselves in time to go home). Plays are usually staged with an interval between the third and fourth acts.

Actions like "speech", "entry" and "exit" are really in the category of the lexical terminals which a scanner (in the person of a member of the audience) would recognize as key symbols while watching a play. So one description of such a staged play might be on the lines of

```
Play   =  Act Act Act "interval" Act Act .
Act    =  Scene { Scene } .
Scene  =  { "speech" } "entry" { Action } .
Action =  "speech" | "entry" | "exit" | "death" | "gesticulation" .
```

This does not require all the actors to leave at the end of any scene (sometimes this does not happen in real life, either). We could try to get this effect by writing

```
Scene  =  { "speech" } "entry" { Action } { "exit" } .
```

but note that this context-free grammar cannot force as many actors to leave as entered - in computer language terms the reader should recognize this as the same problem as being unable to specify that the number of formal and actual parameters to a procedure agree.

Analyse this grammar in detail. If it proves out to be non-LL(1), try to find an equivalent that *is* LL(1), or argue why this should be impossible.

9.3 The effect of the LL(1) conditions on language design

There are some immediate implications which follow from the rules of the last section as regards language design and specification. Alternative right-hand sides for productions are very common; we cannot hope to avoid their use in practice. Let us consider some common situations where problems might arise, and see whether we can ensure that the conditions are met.

Firstly, we should note that we cannot hope to transform every non-LL(1) grammar into an

equivalent LL(1) grammar. To take an extreme example, an ambiguous grammar must have two parse trees for at least one input sentence. If we *really* want to allow this we shall not be able to use a parsing method that is capable of finding only one parse tree, as deterministic parsers must do. We can argue that an ambiguous grammar is of little interest, but the reader should not go away with the impression that it is just a matter of trial and error before an equivalent LL(1) grammar is found for an arbitrary grammar.

Often a combination of substitution and re-factorization will resolve problems. For example, it is almost trivially easy to find a grammar for the problematic language of section 9.1 which satisfies Rule 1. Once we have seen the types of strings the language allows, then we easily see that all we have to do is to find productions that sensibly deal with leading strings of *x*'s, but delay introducing *y* and *z* for as long as possible. This insight leads to productions of the form

```
A  →  xA  |  C
C  →  y   |  z
```

Productions with alternatives are often found in specifying the kinds of *Statement* that a programming language may have. Rule 1 suggests that if we wish to parse programs in such a language by using LL(1) techniques we should design the language so that each statement type begins with a different reserved keyword. This is what is attempted in several languages, but it is not always convenient, and we may have to get round the problem by factorizing the grammar differently.

As another example, if we were to extend the language of section 8.7 we might contemplate introducing **REPEAT** loops in one of two forms

```
RepeatStatement     =    "REPEAT"  StatementSequence  "UNTIL"  Condition
                       |  "REPEAT"  StatementSequence  "FOREVER" .
```

Both of these start with the reserved word REPEAT. However, if we define

```
RepeatStatement     =  "REPEAT"  StatementSequence  TailRepeatStatement .
TailRepeatStatement =  "UNTIL"  Condition  |  "FOREVER" .
```

parsing can proceed quite happily. Another case which probably comes to mind is provided by the statements

```
Statement           =  IfStatement  |  OtherStatement .
IfStatement         =   "IF"  Condition  "THEN"  Statement
                       | "IF"  Condition  "THEN"  Statement  "ELSE"  Statement .
```

Factorization on the same lines as for the REPEAT loop is less successful. We might be tempted to try

```
Statement           =  IfStatement  |  OtherStatement .              (1, 2)
IfStatement         =  "IF"  Condition  "THEN"  Statement  IfTail .   (3)
IfTail              =  "ELSE"  Statement  |  ε .                     (4, 5)
```

but then we run foul of Rule 2. The production for *IfTail* is nullable; a little reflection shows that

```
FIRST("ELSE" Statement) = { "ELSE" }
```

while to compute FOLLOW(*IfTail*) we consider the production (3) (which is where *IfTail* appears on the right side), and obtain

```
FOLLOW(IfTail)   = FOLLOW(IfStatement)     (production 3)
                 = FOLLOW(Statement)       (production 1)
```

which clearly includes ELSE.

The reader will recognize this as the "dangling else" problem again. We have already remarked that we can find ways of expressing this construct unambiguously; but in fact the more usual solution is just to impose the semantic meaning that the `ELSE` is attached to the most recent unmatched `THEN`, which, as the reader will discover, is handled trivially easily by a recursive descent parser. (Semantic resolution is quite often used to handle tricky points in recursive descent parsers, as we shall see.)

Perhaps not quite so obviously, Rule 1 eliminates the possibility of using left recursion to specify syntax. This is a very common way of expressing a repeated pattern of symbols in BNF. For example, the two productions

$$A \rightarrow B \mid AB$$

describe the set of sequences *B* , *BB* , *BBB* ... . Their use is now ruled out by Rule 1, because

$$\text{FIRST}(A_1) = \text{FIRST(B)}$$
$$\text{FIRST}(A_2) = \text{FIRST}(AB) = \text{FIRST}(A) = \text{FIRST}(B) \cup \text{FIRST}(AB)$$
$$\text{FIRST}(A_1) \cap \text{FIRST}(A_2) \neq \emptyset$$

Direct left recursion can be avoided by using right recursion. Care must be taken, as sometimes the resulting grammar is still unsuitable. For example, the productions above are equivalent to

$$A \rightarrow B \mid BA$$

but this still more clearly violates Rule 1. In this case, the secret lies in deliberately introducing extra non- terminals. A non-terminal which admits to left recursive productions will in general have two alternative productions, of the form

$$A \rightarrow AX \mid Y$$

By expansion we can see that this leads to sentential forms like

$$Y \text{ , } YX \text{ , } YXX \text{ , } YXXX$$

and these can easily be derived by the equivalent grammar

$$A \rightarrow YZ$$
$$Z \rightarrow \varepsilon \mid XZ$$

The example given earlier is easily dealt with in this way by writing $X = Y = B$, that is

$$A \rightarrow BZ$$
$$Z \rightarrow \varepsilon \mid BZ$$

The reader might complain that the limitation on two alternatives for *A* is too severe. This is not really true, as suitable factorization can allow *X* and *Y* to have alternatives, none of which start with *A*. For example, the set of productions

$$A \rightarrow Ab \mid Ac \mid d \mid e$$

can obviously be recast as

$A \rightarrow AX \mid Y$
$X \rightarrow b \mid c$
$Y \rightarrow d \mid e$

(Indirect left recursion, for example

$A \rightarrow B$
$B \rightarrow C \ldots$
$C \rightarrow A \ldots$

is harder to handle, and is, fortunately, not very common in practice.)

This might not be quite as useful as it first appears. For example, the problem with

```
Expression      =  Expression  "-"  Term  |  Term .
```

can readily be removed by using right recursion

```
Expression      =  Term  RestExpression .
RestExpression  =  ε  |  "-"  Term  RestExpression .
```

but this may have the side-effect of altering the implied order of evaluation of an *Expression*. For example, adding the productions

```
Term            =  "x"  |  "y"  |  "z" .
```

to the above would mean that with the former production for *Expression*, a string of the form *x* - *y* - *z* would be evaluated as (*x* - *y*) - *z*. With the latter production it might be evaluated as *x* - (*y* - *z*), which would result in a very different answer (unless *z* were zero).

The way to handle this situation would be to write the parsing algorithms to use iteration, as introduced earlier, for example

```
Expression      =  Term {  "-"  Term } .
```

Although this is merely another way of expressing the right recursive productions used above, it may be easier for the reader to follow. It carries the further advantage of more easily retaining the left associativity which the "-" terminal normally implies.

It might be tempting to try to use such iteration to remove all the problems associated with recursion. Again, care must be taken, since this action often implies that ε-productions either explicitly or implicitly enter the grammar. For example, the construction

$A \rightarrow \{ B \}$

actually implies, and can be written

$A \rightarrow \varepsilon \mid B A$

but can only be handled if FIRST(B) ∩ FOLLOW(A) = Ø. The reader might already have realized that all our manipulations to handle *Expression* would come to naught if "-" could follow *Expression* in other productions of the grammar.

Exercises

9.6 Determine the FIRST and FOLLOW sets for the following non-terminals of the grammar defined in various ways in section 8.7, and comment on which formulations may be parsed using LL(1) techniques.

Block
ConstDeclarations
VarDeclarations
Statement
Expression
Factor
Term

9.7 What are the semantic implications of using the productions suggested in section 8.4 for the `IF ... THEN` and `IF ... THEN ... ELSE` statements?

9.8 Whether to regard the semicolon as a separator or as a terminator has been a matter of some controversy. Do we need semicolons at all in a language like the one suggested in section 8.7? Try to write productions for a version of the language where they are simply omitted, and check whether the grammar you produce satisfies the LL(1) conditions. If it does not, try to modify the grammar until it does satisfy these conditions.

9.9 A close look at the syntax of Pascal, Modula-2 or the language of section 8.7 shows that an ε-production is allowed for *Statement*. Can you think of any reasons at all why one should not simply forbid empty statements?

9.10 Write down a set of productions that describes the form that `REAL` literal constants may assume in Pascal, and check to see whether they satisfy the LL(1) conditions. Repeat the exercise for `REAL` literal constants in Modula-2 and for `float` literals in C++ (surprisingly, perhaps, the grammars are different).

9.11 In a language like Modula-2 or Pascal there are two classes of statements that start with identifiers, namely assignment statements and procedure calls. Is it possible to find a grammar that allows this potential LL(1) conflict to be resolved? Does the problem arise in C++?

9.12 A full description of C or C++ is not possible with an LL(1) grammar. How large a subset of these languages could one describe with an LL(1) grammar?

9.13 C++ and Modula-2 are actually fairly close in many respects - both are imperative, both have the same sorts of statements, both allow user defined data structures, both have functions and procedures. What features of C++ make description in terms of LL(1) grammars difficult or impossible, and is it easier or more difficult to describe the corresponding features in Modula-2? Why?

9.14 Why do you suppose C++ has so many levels of precedence and the rules it does have for associativity? What do they offer to a programmer that Modula-2 might appear to withhold? Does Modula-2 really withhold these features?

9.15 Do you suppose there may be any correlation between the difficulty of writing a grammar for a

language (which programmers do not usually try to do) and learning to write programs in that language (which programmers often do)?

Further reading

Good treatments of the material in this chapter may be found at a comprehensible level in the books by Wirth (1976b, 1996), Welsh and McKeag (1980), Hunter (1985), Gough (1988), Rechenberg and Mössenböck (1989), and Tremblay and Sorenson (1985). Pittman and Peters (1992) have a good discussion of what can be done to transform non-LL(k) grammars into LL(k) ones.

Algorithms exist for the detection and elimination of useless productions. For a discussion of these the reader is referred to the books by Gough (1988), Rechenberg and Mössenböck (1989), and Tremblay and Sorenson (1985).

Our treatment of the LL(1) conditions may have left the reader wondering whether the process of checking them - especially the second one - ever converges for a grammar with anything like the number of productions needed to describe a real programming language. In fact, a little thought should suggest that, even though the number of sentences which they can generate might be infinite, convergence should be guaranteed, since the number of productions is finite. The process of checking the LL(k) conditions can be automated, and algorithms for doing this and further discussion of convergence can be found in the books mentioned above.

10 PARSER AND SCANNER CONSTRUCTION

In this chapter we aim to show how parsers and scanners may be synthesized once appropriate grammars have been written. Our treatment covers the manual construction of these important components of the translation process, as well as an introduction to the use of software tools that help automate the process.

10.1 Construction of simple recursive descent parsers

For the kinds of language that satisfy the rules discussed in the last chapter, parser construction turns out to be remarkably easy. The syntax of these languages is governed by production rules of the form

non-terminal → allowable string

where the allowable string is a concatenation derived from

- the basic symbols or terminals of the language
- other non-terminals
- the actions of meta-symbols such as { }, [ ], and | .

We express the effect of applying each production by writing a procedure (or *void function* in C++ terminology) to which we give the name of the non-terminal that appears on its left side. The purpose of this routine is to analyse a sequence of symbols, which will be supplied on request from a suitable scanner (lexical analyser), and to verify that it is of the correct form, reporting errors if it is not. To ensure consistency, the routine corresponding to any non-terminal *S*:

- may assume that it has been called *after* some (globally accessible) variable *Sym* has been found to contain one of the terminals in FIRST(*S*).

- will then parse a complete sequence of terminals which can be derived from *S*, reporting an error if no such sequence is found. (In doing this it may have to call on similar routines to handle sub-sequences.)

- will relinquish parsing after leaving *Sym* with the first terminal that it finds which cannot be derived from *S*, that is to say, a member of the set FOLLOW(*S*).

The shell of each parsing routine is thus

```
PROCEDURE S;
(* S  →  string *)
  BEGIN
    (* we assert Sym  ∈  FIRST(S) *)
    Parse(string)
    (* we assert  Sym  ∈  FOLLOW(S) *)
  END S;
```

where the transformation *Parse(string)* is governed by the following rules:

(a) If the production yields a single terminal, then the action of *Parse* is to report an error if an unexpected terminal is detected, or (more optimistically) to accept it, and then to scan to the next symbol.

```
Parse (terminal) →
  IF IsExpected(terminal)
    THEN Get(Sym)
    ELSE ReportError
  END
```

(b) If we are dealing with a "single" production (that is, one of the form *A* = *B*), then the action of *Parse* is a simple invocation of the corresponding routine

```
Parse(SingleProduction A)  →  B
```

This is a rather trivial case, just mentioned here for completeness. Single productions do not really need special mention, except where they arise in the treatment of longer strings, as discussed below.

(c) If the production allows a number of alternative forms, then the action can be expressed as a selection

```
Parse (α1  |  α2  |  ... αn ) →
  CASE Sym OF
    FIRST(α1 ) : Parse(α1 );
    FIRST(α2 ) : Parse(α2 );
    ......
    FIRST(αn ) : Parse(αn )
  END
```

in which we see immediately the relevance of Rule 1. In fact we can go further to see the relevance of Rule 2, for to the above we should add the action to be taken if one of the alternatives of *Parse* is empty. Here we do nothing to advance *Sym* - an action which must leave *Sym*, as we have seen, as one of the set FOLLOW(*S*) - so that we may augment the above *in this case* as

```
Parse (α1  |  α2  | ... αn  |  ε ) →
  CASE Sym OF
    FIRST(α1 ) : Parse(α1 );
    FIRST(α2 ) : Parse(α2 );
    ......
    FIRST(αn ) : Parse(αn );
    FOLLOW(S)  :  (* do nothing *)
    ELSE ReportError
  END
```

(d) If the production allows for a nullable option, the transformation involves a decision

```
Parse ( [ α ] ) →
  IF Sym ∈ FIRST(α) THEN Parse(α) END
```

(e) If the production allows for possible repetition, the transformation involves a loop, often of the form

```
Parse ( { α } ) →
  WHILE Sym ∈ FIRST(α) DO Parse(α) END
```

Note the importance of Rule 2 here again. Some repetitions are of the form

```
S  →  α  { α }
```

which transforms to

```
Parse(α);  WHILE Sym ∈ FIRST(α) DO Parse(α)  END
```

On occasions this may be better written

```
REPEAT  Parse(α) UNTIL Sym ∉ FIRST(α)
```

(f) Very often, the production generates a sequence of terminal and non-terminals. The action is then a sequence derived from (a) and (b), namely

```
Parse (α1 α2 ... αn ) →
  Parse(α1 ); Parse(α2 ); ... Parse(αn )
```

10.2 Case studies

To illustrate these ideas further, let us consider some concrete examples.

The first involves a rather simple grammar, chosen to illustrate the various options discussed above.

```
G = { N , T , S , P }
N = { A , B , C , D }
T = { "("  ,  ")"  ,  "+"  ,  "a"  ,  "["  ,  "]"  ,  "." }
S = A
P =
    A  →  B  "."
    B  →  [  "a"   |   "("  C  ")"   |   "["  B  "]"  ]
    C  →  B D
    D  →  { "+"  B }
```

We first check that this language satisfies the requirements for LL(1) parsing. We can easily see that Rule 1 is satisfied. As before, in order to apply our rules more easily we first rewrite the productions to eliminate the EBNF metasymbols:

```
A  →  B  "."                                          (1)
B  →  "a"   |   "("  C  ")"   |   "["  B  "]"  | ε   (2, 3, 4, 5)
C  →  B D                                             (6)
D  →  "+"  B  D  | ε                                  (7, 8)
```

The only productions for which there are alternatives are those for *B* and *D*, and each non-nullable alternative starts with a different terminal. However, we must continue to check Rule 2. We note that *B* and *D* can both generate the null string. We readily compute

FIRST(*B*) = { "*a*" , "(" , "[" }
FIRST(*D*) = { "+" }

The computation of the FOLLOW sets is a little trickier. We need to compute FOLLOW(*B*) and FOLLOW(*D*).

For FOLLOW(*D*) we use the rules of section 9.2. We check productions that generate strings of the form $\alpha D \zeta$. These are the ones for *C* (6) and for *D* (7). Both of these have *D* as their rightmost symbol; (7) in fact tells us nothing of interest, and we are lead to the result that

FOLLOW(*D*) = FOLLOW(*C*) = { ")" }.

(FOLLOW(*C*) is determined by looking at production (3)).

For FOLLOW(*B*) we check productions that generate strings of the form $\alpha B \zeta$. These are the ones for *A* (1) and *C* (6), the third alternative for *B* itself (4), and the first alternative for *D* (7). This

seems to indicate that

FOLLOW(*B*) = { "." , "]" } ∪ FIRST(*D*) = { "." , "]" , "+" }

We must be more careful. Since the production for *D* can generate a null string, we must augment FOLLOW(*B*) by FOLLOW(*C*) to give

FOLLOW(*B*) = { "." , "]" , "+" } ∪ { ")" } = { "." , "]" , "+" , ")" }

Since FIRST(*B*) ∩ FOLLOW(*B*) = Ø and FIRST(*D*) ∩ FOLLOW(*D*) = Ø, Rule 2 is satisfied for both the non-terminals that generate alternatives, both of which are nullable.

A C++ program for a parser follows. The terminals of the language are all single characters, so that we do not have to make any special arrangements for character handling (a simple `getchar` function call suffices) or for lexical analysis.

The reader should note that, because the grammar is strictly LL(1), the function that parses the non-terminal *B* may discriminate between the genuine followers of *B* (thereby effectively recognizing where the ε-production needs to be applied) and any spurious followers of *B* (which would signal a gross error in the parsing process).

```
// Simple Recursive Descent Parser for the language defined by the grammar
//         G = { N , T , S , P }
//         N = { A , B , C , D }
//         T = { "(" , ")" , "+" , "a" , "[" , "]" , "." }
//         S = A
//         P =
//             A  =  B "." .
//             B  =  "a"   |   "(" C ")"   |   "[" B "]"   |   .
//             C  =  B D .
//             D  =  { "+" B } .
// P.D. Terry,  Rhodes University, 1996

#include <stdio.h>
#include <stdlib.h>

char sym;  // Source token

void getsym(void)
{ sym = getchar(); }

void accept(char expectedterminal, char *errormessage)
{ if (sym != expectedterminal) { puts(errormessage); exit(1); }
  getsym();
}

void A(void);  // prototypes
void B(void);
void C(void);
void D(void);

void A(void)
//  A = B "." .
{ B(); accept('.', " Error - '.' expected"); }

void B(void)
//  B  =  "a"   |   "(" C ")"   |   "[" B "]"   |   .
{ switch (sym)
  { case 'a':
      getsym(); break;
    case '(':
      getsym(); C(); accept(')', " Error - ')' expected"); break;
    case '[':
      getsym(); B(); accept(']', " Error - ']' expected"); break;
    case ')':
    case ']':
    case '+':
    case '.':
      break;  // no action for followers of B
    default:
```

```
      printf("Unknown symbol\n"); exit(1);
  }
}

void C(void)
//  C = B D .
{ B(); D(); }

void D(void)
//  D  =  { "+" B } .
{ while (sym == '+') { getsym(); B(); } }

void main()
{ sym = getchar(); A();
  printf("Successful\n");
}
```

Some care may have to be taken with the relative ordering of the declaration of the functions, which in this example, and in general, are recursive in nature. (These problems do not occur if the functions have "prototypes" like those illustrated here.)

It should now be clear why this method of parsing is called *Recursive Descent*, and that such parsers are most easily implemented in languages which directly support recursive programming. Languages like Modula-2 and C++ are all very well suited to the task, although they each have their own particular strengths and weaknesses. For example, in Modula-2 one can take advantage of other organizational strategies, such as the use of nested procedures (which are not permitted in C or C++), and the very tight control offered by encapsulating a parser in a module with a very thin interface (only the routine for the goal symbol need be exported), while in C++ one can take advantage of OOP facilities (both to encapsulate the parser with a thin public interface, and to create hierarchies of specialized parser classes).

A little reflection shows that one can often combine routines (this corresponds to reducing the number of productions used to define the grammar). While this may produce a shorter program, precautions must be taken to ensure that the grammars, and any implicit semantic overtones, are truly equivalent. An equivalent grammar to the above one is

```
G = { N , T , S , P }
N = { A , B }
T = { "("  ,  ")"  ,  "+"  ,  "a"  ,  "["  ,  "]"  ,  "." }
S = A
P =
    A  ⟶  B "."                                                    (1)
    B  ⟶  "a"   |   "("  B  { "+"  B }  ")"   |   "["  B  "]"   |  ε (2, 3, 4, 5)
```

leading to a parser

```
// Simple Recursive Descent Parser for the same language
// using an equivalent but different grammar
// P.D. Terry,  Rhodes University, 1996

#include <stdio.h>
#include <stdlib.h>

char sym;  // Source token

void getsym(void)
{ sym = getchar(); }

void accept(char expectedterminal, char *errormessage)
{ if (sym != expectedterminal) { puts(errormessage); exit(1); }
  getsym();
}

void B(void)
//  B = "a"  |  "(" B { "+" B } ")"  |  "[" B "]"  |   .
{ switch (sym)
  { case 'a':
      getsym(); break;
    case '(':
      getsym(); B(); while (sym == '+') { getsym(); B(); }
```

```
      accept(')', " Error - ')' expected"); break;
    case '[':
      getsym(); B(); accept(']', " Error - ']' expected"); break;
    case ')':
    case ']':
    case '+':
    case '.':
      break;    // no action for followers of B
    default:
      printf("Unknown symbol\n"); exit(1);
  }
}

void A(void)
//  A = B "." .
{ B(); accept('.', " Error - '.' expected"); }

void main(void)
{ sym = getchar(); A();
  printf("Successful\n");
}
```

Although recursive descent parsers are eminently suitable for handling languages which satisfy the LL(1) conditions, they may often be used, perhaps with simple modifications, to handle languages which, strictly, do not satisfy these conditions. The classic example of a situation like this is provided by the `IF ... THEN ... ELSE` statement. Suppose we have a language in which statements are defined by

```
Statement     =  IfStatement  |  OtherStatement .
IfStatement   =  "IF"  Condition  "THEN"  Statement [ "ELSE" Statement  ] .
```

which, as we have already discussed, is actually ambiguous as it stands. A grammar defined like this is easily parsed deterministically with code like

```
void Statement(void); // prototype

void OtherStatement(void);
// handle parsing of other statement - not necessary to show this here

void IfStatement(void)
{ getsym(); Condition();
  accept(thensym, " Error - 'THEN' expected");
  Statement();
  if (sym == elsesym) { getsym(); Statement(); }
}

void Statement(void)
{ switch(sym)
  { case ifsym : IfStatement(); break;
    default :    OtherStatement(); break;
  }
}
```

The reader who cares to trace the function calls for an input sentence of the form

```
IF  Condition  THEN  IF  Condition  THEN  OtherStatement  ELSE  OtherStatement
```

will note that this parser has the effect of recognizing and handling an `ELSE` clause as soon as it can - effectively forcing an *ad hoc* resolution of the ambiguity by coupling each `ELSE` to the closest unmatched `THEN`. Indeed, it would be far more difficult to design a parser that implemented the other possible disambiguating rule - no wonder that the semantics of this statement are those which correspond to the solution that becomes easy to parse!

As a further example of applying the LL(1) rules and considering the corresponding parsers, consider how one might try to describe variable designators of the kind found in many languages to denote elements of record structures and arrays, possibly in combination, for example `A[B.C.D]`. One set of productions that describes some (although by no means all) of these constructions might appear to be:

```
Designator     =  identifier  Qualifier .                (1)
Qualifier      =  Subscript | FieldSpecifier .           (2, 3)
Subscript      =   "["  Designator  "]" | ε  .           (4, 5)
FieldSpecifier =   "."  Designator  | ε .                (6, 7)
```

This grammar is not LL(1), although it may be at first difficult to see this. The production for *Qualifier* has alternatives, and to check Rule 1 for productions 2 and 3 we need to consider FIRST($Qualifier_1$) and FIRST($Qualifier_2$). At first it appears obvious that

FIRST($Qualifier_1$) = FIRST(*Subscript*) = { "[" }

but we must be more careful. *Subscript* is nullable, so to find FIRST($Qualifier_1$) we must augment this singleton set with FOLLOW(*Subscript*). The calculation of this requires that we find productions with *Subscript* on the right side - there is only one of these, production (2). From this we see that FOLLOW(*Subscript*) = FOLLOW(*Qualifier*), which from production (1) is FOLLOW(*Designator*). To determine FOLLOW(*Designator*) we must examine productions (4) and (6). Only the first of these contributes anything, namely { "]" }. Thus we eventually conclude that

FIRST($Qualifier_1$) = { "[", "]" }.

Similarly, the obvious conclusion that

FIRST($Qualifier_2$) = FIRST(*FieldSpecifier*) = { "." }

is also too naïve (since *FieldSpecifier* is also nullable); a calculation on the same lines leads to the result that

FIRST($Qualifier_2$) = { "." , "]" }

Rule 1 is thus broken; the grammar is not LL(1).

The reader will complain that this is ridiculous. Indeed, rewriting the grammar in the form

```
Designator     =  identifier Qualifier .                 (1)
Qualifier      =  Subscript  |  FieldSpecifier | ε .     (2, 3, 4)
Subscript      =   "[" Designator "]" .                  (5)
FieldSpecifier =   "." Designator .                      (6)
```

leads to no such transgressions of Rule 1, or, indeed of Rule 2 (readers should verify this to their own satisfaction). Once again, a recursive descent parser is easily written:

```
void Designator(void); // prototype

void Subscript(void)
{ getsym(); Designator(); accept(rbracket, " Error - ']' expected"); }

void FieldSpecifier(void)
{ getsym(); Designator(); }

void Qualifier(void)
{ switch(sym)
  { case lbracket : Subscript(); break;
    case period   : FieldSpecifier(); break;
    case rbracket : break; // FOLLOW(Qualifier) is empty
    default :       printf("Unknown symbol\n"); exit(1);
  }
}

void Designator(void)
```

```
{ accept(identifier, " Error - identifier expected");
  Qualifier();
}
```

In this case there is an easy, if not even obvious way to repair the grammar, and to develop the parser. However, a more realistic version of this problem leads to a situation that cannot as easily be resolved. In Modula-2 a *Designator* is better described by the productions

```
Designator          =  QualifiedIdentifier  {  Selector  } .
QualifiedIdentifier =  identifier  { "."  identifier } .
Selector            =  "."  identifier  |  "["  Expression  "]"  | "^" .
```

It is left as an exercise to demonstrate that this is not LL(1). It is left as a harder exercise to come to a formal conclusion that one cannot find an LL(1) grammar that describes *Designator* unambiguously. The underlying reason is that "." is used in one context to separate a module identifier from the identifier that it qualifies (as in `Scanner.SYM`) and in a different context to separate a record identifier from a field identifier (as in `SYM.Name`). When these are combined (as in `Scanner.SYM.Name`) the problem becomes more obvious.

The reader may have wondered at the fact that the parsing methods we have advocated all look "ahead", and never seem to make use of what has already been achieved, that is, of information which has become embedded in the previous history of the parse. All LL(1) grammars are, of course, context-free, yet we pointed out in Chapter 8 that there are features of programming languages which cannot be specified in a context-free grammar (such as the requirement that variables must be declared before use, and that expressions may only be formed when terms and factors are of the correct types). In practice, of course, a parser is usually combined with a semantic analyser; in a sense some of the past history of the parse is recorded in such devices as symbol tables which the semantic analysis needs to maintain. The example given here is not as serious as it may at first appear. By making recourse to the symbol table, a Modula-2 compiler will be able to resolve the potential ambiguity in a static semantic way (rather than in an *ad hoc* syntactic way as is done for the "dangling else" situation).

Exercises

10.1 Check the LL(1) conditions for the equivalent grammar used in the second of the programs above.

10.2 Rework Exercise 10.1 by checking the director sets for the productions.

10.3 Suppose we wished the language in the previous example to be such that spaces in the input file were irrelevant. How could this be done?

10.4 In section 8.4 an unambiguous set of productions was given for the `IF ... THEN ... ELSE` statement. Is the corresponding grammar LL(1)? Whatever the outcome, can you construct a recursive descent parser to handle such a formulation of the grammar?

10.3 Syntax error detection and recovery

Up to this point our parsers have been content merely to stop when a syntactic error is detected. In the case of a real compiler this is probably unacceptable. However, if we modify the parser as given

above so as simply not to stop after detecting an error, the result is likely to be chaotic. The analysis process will quickly get out of step with the sequence of symbols being scanned, and in all likelihood will then report a plethora of spurious errors.

One useful feature of the compilation technique we are using is that the parser can detect a syntactically incorrect structure after being presented with its first "unexpected" terminal. This will not necessarily be at the point where the error really occurred. For example, in parsing the sequence

```
BEGIN IF A > 6 DO B := 2; C := 5 END END
```

we could hope for a sensible error message when `DO` is found where `THEN` is expected. Even if parsing does not get out of step, we would get a less helpful message when the second `END` is found - the compiler can have little idea where the missing `BEGIN` should have been.

A production quality compiler should aim to issue appropriate diagnostic messages for all the "genuine" errors, and for as few "spurious" errors as possible. This is only possible if it can make some likely assumption about the nature of each error and the probable intention of the author, or if it skips over some part of the malformed text, or both. Various approaches may be made to handling the problem. Some compilers go so far as to try to correct the error, and continue to produce object code for the program. Error correction is a little dangerous, except in some trivial cases, and we shall discuss it no further here. Many systems confine themselves to attempting **error recovery**, which is the term used to describe the process of simply trying to get the parser back into step with the source code presented to it. The art of doing this for hand-crafted compilers is rather intricate, and relies on a mixture of fairly well defined methods and intuitive experience, both with the language being compiled, and with the class of user of the same.

Since recursive descent parsers are constructed as a set of routines, each of which tackles a sub-goal on behalf of its caller, a fairly obvious place to try to regain lost synchronization is at the entry to and exit from these routines, where the effects of getting out of step can be confined to examining a small range of known FIRST and FOLLOW symbols. To enforce synchronization at the entry to the routine for a non-terminal *S* we might try to employ a strategy like

```
IF Sym ∉ FIRST(S) THEN
  ReportError; SkipTo(FIRST(S))
END
```

where *SkipTo* is an operation which simply calls on the scanner until it returns a value for *Sym* that is a member of FIRST(*S*). Unfortunately this is not quite adequate - if the leading terminal has been omitted we might then skip over symbols that should be processed later, by the routine which called *S*.

At the exit from *S*, we have postulated that *Sym* should be a member of FOLLOW(*S*). This set may not be known to *S*, but it should be known to the routine which calls *S*, so that it may conveniently be passed to *S* as a parameter. This suggests that we might employ a strategy like

```
IF Sym ∉ FOLLOW(S) THEN
  ReportError; SkipTo(FOLLOW(S))
END
```

The use of FOLLOW(*S*) also allows us to avoid the danger mentioned earlier of skipping too far at routine entry, by employing a strategy like

```
IF Sym ∉ FIRST(S) THEN
  ReportError; SkipTo(FIRST(S) | FOLLOW(S))
END;
IF SYM.Sym ∈ FIRST(S) THEN
```

```
        Parse(S);
        IF SYM.Sym ∉ FOLLOW(S) THEN
          ReportError; SkipTo(FOLLOW(S))
        END
      END
```

Although the FOLLOW set for a non-terminal is quite easy to determine, the legitimate follower may itself have been omitted, and this may lead to too many symbols being skipped at routine exit. To prevent this, a parser using this approach usually passes to each sub-parser a *Followers* parameter, which is constructed so as to include

- the minimally correct set FOLLOW(*S*), augmented by
- symbols that have already been passed as *Followers* to the calling routine (that is, later followers), and also
- so-called **beacon symbols**, which are on no account to be passed over, even though their presence would be quite out of context. In this way the parser can often avoid skipping large sections of possibly important code.

On return from sub-parser *S* we can then be fairly certain that *Sym* contains a terminal which was either expected (if it is in FOLLOW(*S*)), or can be used to regain synchronization (if it is one of the beacons, or is in FOLLOW(*Caller(S)*). The caller may need to make a further test to see which of these conditions has arisen.

In languages like Modula-2 and Pascal, where set operations are directly supported, implementing this scheme is straightforward. C++ does not have "built-in" set types. Their implementation in terms of a template class is easily achieved, and operator overloading can be put to good effect. An interface to such a class, suited to our applications in this text, can be defined as follows

```
template <int maxElem>
class Set {                          // { 0 .. maxElem }
  public:
    Set();                           // Construct { }
    Set(int e1);                     // Construct { e1 }
    Set(int e1, int e2);             // Construct { e1, e2 }
    Set(int e1, int e2, int e3);     // Construct { e1, e2, e3 }
    Set(int n, int e[]);             // Construct { e[0] .. e[n-1] }
    void incl(int e);                // Include e
    void excl(int e);                // Exclude e
    int memb(int e);                 // Test membership for e
    Set operator + (const Set &s)    // Union with s               (OR)
    Set operator * (const Set &s)    // Intersection with s        (AND)
    Set operator - (const Set &s)    // Difference with s
    Set operator / (const Set &s)    // Symmetric difference with s (XOR)
  private:
    unsigned char bits[(maxElem + 8) / 8];
    int length;
    int wrd(int i);
    int bitmask(int i);
    void clear();
};
```

The implementation is realized by treating a large set as an array of small bitsets; full details of this can be found in the source code supplied on the accompanying diskette and in Appendix B.

Syntax error recovery is then conveniently implemented by defining functions on the lines of

```
typedef Set<lastDefinedSym> symset;

void accept(symtypes expected, int errorcode)
{ if (Sym == expected) getsym(); else reporterror(errorcode); }

void test(symset allowed, symset beacons, int errorcode)
{ if (allowed.memb(Sym)) return;
```

```
  reporterror(errorcode);
  symset stopset = allowed + beacons;
  while (!stopset.memb(Sym)) getsym();
}
```

where we note that the amended `accept` routine does not try to regain synchronization in any way. The way in which these functions could be used is exemplified in a routine for handling variable declarations for Clang:

```
void VarDeclarations(symset followers);
// VarDeclarations = "VAR" OneVar { "," OneVar } ";" .
{ getsym();                              // accept "var"
  test(symset(identifier), followers, 6); // FIRST(OneVar)
  if (Sym == identifier)                 // we are in step
  { OneVar(symset(comma, semicolon) + followers);
    while (Sym == comma)                 // more variables follow
    { getsym(); OneVar(symset(comma, semicolon) + followers); }
    accept(semicolon, 2);
    test(followers, symset(), 34);
  }
}
```

The `followers` passed to `VarDeclarations` should include as "beacons" the elements of FIRST(*Statement*) - symbols which could start a *Statement* (in case `BEGIN` was omitted) - and the symbol which could follow a *Block* (period, and end-of-file). Hence, calling `VarDeclarations` might be done from within `Block` on the lines of

```
if (Sym == varsym)
  VarDeclarations(FirstBlock + FirstStatement + followers);
```

Too rigorous an adoption of this scheme will result in some spurious errors, as well as an efficiency loss resulting from all the set constructions that are needed. In hand-crafted parsers the ideas are often adapted somewhat. As mentioned earlier, one gains from experience when dealing with learners, and some concession to likely mistakes is, perhaps, a good thing. For example, beginners are likely to confuse operators like ":=", "=" and "==", and also `THEN` and `DO` after `IF`, and these may call for special treatment. As an example of such an adaptation, consider the following variation on the above code, where the parser will, in effect, handle variable declarations in which the separating commas have been omitted. This is strategically a good idea - variable declarations that are not properly processed are likely to lead to severe difficulties in handling later stages of a compilation.

```
void VarDeclarations(symset followers);
// VarDeclarations = "VAR" OneVar { "," OneVar } ";" .
{ getsym()                                  // accept "var"
  test(symset(identifier), followers, 6);   // FIRST(OneVar)
  if (Sym == identifier)                    // we are in step
  { OneVar(symset(comma, semicolon) + followers);
    while (Sym == comma || Sym == identifier) // only comma is legal
    { accept(comma), 31); OneVar(symset(comma, semicolon) + followers); }
    accept(semicolon, 2);
    test(followers, symset(), 34);
  }
}
```

Clearly it is impossible to recover from all possible contortions of code, but one should guard against the cardinal sins of not reporting errors when they are present, or of collapsing completely when trying to recover from an error, either by giving up prematurely, or by getting the parser caught in an infinite loop reporting the same error.

Exercises

10.5 Extend the parsers developed in section 10.2 to incorporate error recovery.

10.6 Investigate the efficacy of the scheme suggested for parsing variable declarations, by tracing the way in which parsing would proceed for incorrect source code such as the following:

```
VAR A, B C , , D; E, F;
```

Further reading

Error recovery is an extensive topic, and we shall have more to say on it in later chapters. Good treatments of the material of this section may be found in the books by Welsh and McKeag (1980), Wirth (1976b), Gough (1988) and Elder (1994). A much higher level treatment is given by Backhouse (1979), while a rather simplified version is given by Brinch Hansen (1983, 1985). Papers by Pemberton (1980) and by Topor (1982), Stirling (1985) and Grosch (1990b) are also worth exploring, as is the bibliographical review article by van den Bosch (1992).

10.4 Construction of simple scanners

In a sense, a scanner or lexical analyser may be thought of as just another syntax analyser. It handles a grammar with productions relating non-terminals such as *identifier*, *number* and *Relop* to terminals supplied, in effect, as single characters of the source text. When used in conjunction with a higher level parser a subtle shift in emphasis comes about: there is, in effect, no special goal symbol. Each invocation of the scanner is very much bottom-up rather than top-down; its task ends when it has reduced a string of characters to a token, without preconceived ideas of what that should be. These tokens or non-terminals are then regarded as terminals by the higher level recursive descent parser that analyses the phrase structure of *Block, Statement*, *Expression* and so on.

There are at least five reasons for wishing to decouple the scanner from the main parser:

- The productions involved are usually very simple. Very often they amount to regular expressions, and then a scanner may be programmed without recourse to methods like recursive descent.
- A symbol like an *identifier* is lexically equivalent to a "reserved word"; the distinction may sensibly be made as soon as the basic token has been synthesized.
- The character set may vary from machine to machine, a variation easily isolated in this phase.
- The semantic analysis of a numeric literal constant (deriving the internal representation of its value from the characters) is easily performed in parallel with lexical analysis.
- The scanner can be made responsible for screening out superfluous separators, like blanks and comments, which are rarely of interest in the formulation of the higher level grammar.

In common with the parsing strategy suggested earlier, development of the routine or function responsible for token recognition

- may assume that it is always called *after* some (globally accessible) variable *CH* has been found to contain the next character to be handled in the source

- will then read a complete sequence of characters that form a recognizable token
- will relinquish scanning after leaving *CH* with the first character that does not form part of this token (so as to satisfy the precondition for the next invocation of the scanner).

A scanner is necessarily a top-down parser, and for ease of implementation it is desirable that the productions defining the token grammar also obey the LL(1) rules. However, checking these is much simpler, as token grammars are almost invariably regular, and do not display self-embedding (and thus can be almost always easily be transformed into LL(1) grammars).

There are two main strategies that are employed in scanner construction:

- Rather than being decomposed into a set of recursive routines, simple scanners are often written in an *ad hoc* manner, controlled by a large `CASE` or `switch` statement, since the essential task is one of choosing between a number of tokens, which are sometimes distinguishable on the basis of their initial characters.
- Alternatively, since they usually have to read a number of characters, scanners are often written in the form of a **finite state automaton** (FSA) controlled by a loop, on each iteration of which a single character is absorbed, the machine moving between a number of "states", determined by the character just read. This approach has the advantage that the construction can be formalized in terms of an extensively developed automata theory, leading to algorithms from which scanner generators can be constructed automatically.

A proper discussion of automata theory is beyond the scope of this text, but in the next section we shall demonstrate both approaches to scanner construction by means of some case studies.

10.5 Case studies

To consider a concrete example, suppose that we wish to extend the grammar used for earlier demonstrations into one described in Cocol as follows:

```
COMPILER A
  CHARACTERS
    digit  = "0123456789" .
    letter = "abcdefgefghijklmnopqrstuvwxyz" .
  TOKENS
    number      = digit { digit } .
    identifier  = "a" { letter } .
  PRODUCTIONS
    A = B "." .
    B = identifier | number | "(" C ")" | "(." B ".)" | .
    C = B D .
    D = { "+" B } .
END A.
```

Combinations like `(.` and `.)` are sometimes used to represent the brackets [and] on machines with limited character sets. The tokens we need to be able to recognize are definable by an enumeration:

```
TOKENS = { number, lbrack, lparen, rbrack, rparen, plus, period, identifier }
```

It should be easy to see that these tokens are not uniquely distinguishable on the basis of their leading characters, but it is not difficult to write a set of productions for the token grammar that obeys the LL(1) rules:

```
token  =    digit { digit }   (* number *)
         | "(" [ "." ]        (* lparen, lbrack *)
         | "." [ ")" ]        (* period, rbrack *)
         | ")"                (* rparen *)
         | "+"                (* plus *)
         | "a" { letter }     (* identifier *) .
```

from which an *ad hoc* scanner algorithm follows very easily on the lines of

```
TOKENS FUNCTION GetSym;
(* Precondition:  CH is already available
   Postcondition: CH is left as the character following token *)
  BEGIN
    IgnoreCommentsAndSeparators;
    CASE CH OF
      'a' :
        REPEAT Get(CH) UNTIL CH ∉ {'a' .. 'z'};
        RETURN identifier;
      '0' .. '9' :
        REPEAT Get(CH) UNTIL CH ∉ {'0' .. '9'};
        RETURN number;
      '(' :
        Get(CH);
        IF CH = '.'
          THEN Get(CH); RETURN lbrack
          ELSE RETURN lparen
        END;
      '.' :
        Get(CH);
        IF CH = ')'
          THEN Get(CH); RETURN rbrack
          ELSE RETURN period
        END;
      '+' :
        Get(CH); RETURN plus
      ')' :
        Get(CH); RETURN rparen
      ELSE
        Get(CH); RETURN unknown
    END
  END
```

A characteristic feature of this algorithm - and of most scanners constructed in this way - is that they are governed by a selection statement, within the alternatives of which one frequently finds loops that consume sequences of characters. To illustrate the FSA approach - in which the algorithm is inverted to be governed by a single loop - let us write our grammar in a slightly different way, in which the comments have been placed to reflect the state that a scanner can be thought to possess at the point where a character has just been read.

```
token  =   (* unknown *) digit (* number *) { digit (* number *) }
         | (* unknown *) "(" (* lparen *) [ "." (* lbrack *) ]
         | (* unknown *) "." (* period *) [ ")" (* rbrack *) ]
         | (* unknown *) ")" (* rparen *)
         | (* unknown *) "+" (* plus *)
         | (* unknown *) "a" (* identifier *) { letter (* identifier *)  }
```

Another way of representing this information is in terms of a transition diagram like that shown in Figure 10.1, where, as is more usual, the states have been labelled with small integers, and where the arcs are labelled with the characters whose recognition causes the automaton to move from one state to another.

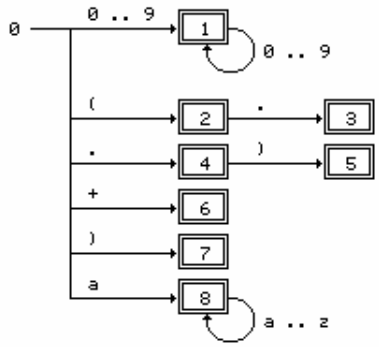


Figure 10.1 A transition diagram for a simple FSA

There are many ways of developing a scanner from these ideas. One approach, using a table-driven scanner, is suggested below. To the set of states suggested by the diagram we add one more, denoted by `finished`, to allow the postcondition to be easily realized.

```
TOKENS FUNCTION GetSym;
(* Preconditions: CH is already available,
                  NextState, Token mappings defined
   Postcondition: CH is left as the character following token *)
  BEGIN
    State := 0;
    WHILE state ≠ finished DO
      LastState := State;
      State := NextState[State, CH];
      Get(CH);
    END;
    RETURN Token[LastState];
  END
```

Here we have made use of various mapping functions, expressed in the form of arrays:

```
Token[s]           is defined to be the token recognized when the machine has reached state s
NextState[s, x]    indicates the transition that must be taken when the machine
                   is currently in state s, and has just recognized character x.
```

For our example, the arrays `Token` and `NextState` would be set up as in the table below. For clarity, the many transitions to the `finished` state have been left blank.

	CH							
State	a	b..z	0..9	(	.	+	)	
0	8		1	2	4	6	7	unknown
1			1					number
2					3			lparen
3								lbrack
4							5	period
5								rbrack
6								plus
7								rparen
8	8	8						identifier
	NextState[State,CH]							Token

A table-driven algorithm is efficient in time, and effectively independent of the token grammar, and thus highly suited to automated construction. However it should not take much imagination to see that it is very hungry and wasteful of storage. A complex scanner might run to dozens of states, and many machines use an ASCII character set, with 256 values. For each character a column would be needed in the matrix, yet most of the entries (as in the example above) would be identical. And although we may have given the impression that this method will always succeed, this is not necessarily so. If the underlying token grammar were not LL(1) it might not be possible to define an unambiguous transition matrix - some entries might appear to require two or more values. In this situation we speak of requiring a **non-deterministic finite automaton** (NDFA) as opposed to the **deterministic finite automaton** (DFA) that we have been considering up until now.

Small wonder that considerable research has been invested in developing variations on this theme. The code below shows one possible variation, for our specimen grammar, in the form of a complete C++ function. In this case it is necessary to have but one static array (denoted by `state0`), initialized so as to map each possible character into a single state.

```
TOKENS getsym(void)
// Preconditions: First character ch has already been read
//                state0[] has been initialized
{ IgnoreCommentsAndSeparators();
  int state = state0[ch];
  while (1)
  { ch = getchar();
    switch (state)
    { case 1 :
        if (!isdigit(ch)) return number;
        break; // state unchanged
      case 2 :
        if (ch = '.') state = 3; else return lparen;
        break;
      case 3 :
        return lbrack;
      case 4 :
        if (ch = ')') state = 5; else return period;
        break;
      case 5 :
        return rbrack;
      case 6 :
        return plus;
      case 7 :
        return rparen;
      case 8 :
        if (!isletter(ch)) return identifier;
        break; // state unchanged
      default :
        return unknown;
    }
  }
}
```

Our scanner algorithms are as yet immature. Earlier we claimed that scanners often incorporated such tasks as the recognition of keywords (which usually resemble identifiers), the evaluation of constant literals, and so on. There are various ways in which these results can be achieved, and in later case studies we shall demonstrate several of them. In the case of the state machine it may be easiest to build up a string that stores all the characters scanned, a task that requires minimal perturbation to the algorithms just discussed. Subsequent processing of this **lexeme** can then be done in an application-specific way. For example, searching for a string in a table of keywords will easily distinguish between keywords and identifiers.

Exercises

10.7 Our scanner algorithms have all had the property that they consume at least one character. Suppose that the initial character could not form part of a token (that is, did not belong to the vocabulary of the language). Would it not be better *not* to consume it?

10.8 Similarly, we have made no provision for the very real possibility that the scanner may not find any characters when it tries to read them, as would happen if it tried to read past the end of the source. Modify the algorithm so that the scanner can recognize this condition, and return a distinctive `eof` token when necessary. Take care to get this correct: the solution may not be as obvious as it at first appears.

10.9 Suppose that our example language was extended to recognize `abs` as a keyword. We could

accomplish this by extending the last part of the transition diagram given earlier to that shown in Figure 10.2.

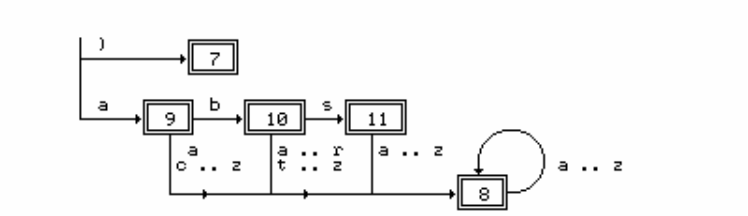


Figure 10.2 Part of a transition diagram for an extended FSA

What corresponding changes would need to be made to the tables needed to drive the parser? In principle one could, of course, handle any number of keywords in a similar fashion. The number of states would grow very rapidly to the stage where manual construction of the table would become very tedious and error-prone.

10.10 How could the C++ code given earlier be modified to handle the extension suggested in Exercise 10.9?

10.11 Suppose our scanner was also required to recognize quoted strings, subject to the common restriction that these should not be allowed to carry across line breaks in the source. How could this be handled? Consider both the extensions that would be needed to the *ad hoc* scanner given earlier, and also to the table driven scanner.

Further reading

Automata theory and the construction of finite state automata are discussed in most texts on compiler construction. A particularly thorough treatment is is to be found in the book by Gough (1988); those by Holub (1990), Watson (1989) and Fischer and LeBlanc (1988, 1991) are also highly readable.

Table driven parsers may also be used to analyse the higher level phrase structure for languages which satisfy the LL(k) conditions. Here, as in the FSA discussed above, and as in the LR parser to be discussed briefly later, the parser itself becomes essentially language independent. The automata have to be more sophisticated, of course. They are known as "push down automata", since they generally need to maintain a stack, so as to be able to handle the self-embedding found in the productions of the grammar. We shall not attempt to discuss such parsers here, but refer the interested reader to the books just mentioned, which all treat the subject thoroughly.

10.6 LR parsing

Although space does not permit of a full description, no modern text on translators would be complete without some mention of so-called **LR(k)** parsing. The terminology here comes from the notion that we scan the input string from **L**eft to right (the L), applying reductions so as to yield a **R**ightmost parse (the R), by looking as far ahead as the next *k* terminals to help decide which production to apply. (In practice *k* is never more than 1, and may be zero.)

The technique is **bottom-up** rather than **top-down**. Starting from the input sentence, and making **reductions**, we aim to end up with the goal symbol. The reduction of a sentential form is achieved by substituting the left side of a production for a string (appearing in the sentential form) which matches the right side, rather than by substituting the right side of a production whose left side appears as a non-terminal in the sentential form.

A bottom-up parsing algorithm might employ a **parse stack**, which contains part of a possible sentential form of terminals and/or non terminals. As we read each terminal from the input string we push it onto the parse stack, and then examine the top elements of this to see whether we can make a reduction. Some terminals may remain on the parse stack quite a long time before they are finally pushed off and discarded. (By way of contrast, a top- down parser can discard the terminals immediately after reading them. Furthermore, a recursive descent parser stores the non-terminal components of the partial sentential form only implicitly, as a chain of as yet uncompleted calls to the routines which handle each non-terminal.)

Perhaps an example will help to make this clearer. Suppose we have a highly simplified (non-LL(1)) grammar for expressions, defined by

```
Goal       =  Expression "." .                 (1)
Expression =  Expression "-" Term  |  Term .   (2, 3)
Term       =  "a"                              (4)
```

and are asked to parse the string "*a - a - a* ." .

The sequence of events could be summarized

Step	Action	Using production	Stack
1	read *a*		*a*
2	reduce	4	*Term*
3	reduce	3	*Expression*
4	read -		*Expression* -
5	read *a*		*Expression* - *a*
6	reduce	4	*Expression* - *Term*
7	reduce	2	*Expression*
8	read -		*Expression* -
9	read *a*		*Expression* - *a*
10	reduce	4	*Expression* - *Term*
11	reduce	2	*Expression*
12	read .		*Expression* .
13	reduce	1	*Goal*

We have reached *Goal* and can conclude that the sentence is valid.

The careful reader may declare that we have cheated! Why did we not use the production *Goal = Expression* when we had reduced the string "*a*" to *Expression* after step 3? To apply a reduction it is, of course necessary that the right side of a production be currently on the parse stack, but this in itself is insufficient. Faced with a choice of right sides which match the top elements on the parse stack, a practical parser will have to employ some strategy, perhaps of looking ahead in the input string, to decide which to apply.

Such parsers are invariably table driven, with the particular strategy at any stage being determined by looking up an entry in a rectangular matrix indexed by two variables, one representing the current "state" of the parse (the position the parser has reached within the productions of the grammar) and the other representing the current "input symbol" (which is one of the terminal or non-terminals of the grammar). The entries in the table specify whether the parser is to *accept* the input string as correct, *reject* as incorrect, *shift* to another state, or *reduce* by applying a particular production. Rather than stack the symbols of the grammar, as was implied by the trace above, the parsing algorithm pushes or pops elements representing states of the parse - a *shift* operation

pushing the newly reached state onto the stack, and a *reduce* operation popping as many elements as there are symbols on the right side of the production being applied. The algorithm can be expressed:

```
BEGIN
  GetSYM(InputSymbol); (* first Sym in sentence *)
  State := 1; Push(State); Parsing := TRUE;
  REPEAT
    Entry := Table[State, InputSymbol];
    CASE Entry.Action OF
      shift:
        State := Entry.NextState; Push(State);
        IF IsTerminal(InputSymbol) THEN
          GetSYM(InputSymbol) (* accept *)
        END
      reduce:
        FOR I := 1 TO Length(Rule[Entry].RightSide) DO Pop END;
        State := Top(Stack);
        InputSymbol := Rule[Entry].LeftSide;
      reject:
        Report(Failure); Parsing := FALSE
      accept:
        Report(Success); Parsing := FALSE
    END
  UNTIL NOT Parsing
END
```

Although the algorithm itself is very simple, construction of the parsing table is considerably more difficult. Here we shall not go into how this is done, but simply note that for the simple example given above the parsing table might appear as follows (we have left the `reject` entries blank for clarity):

				Symbol		
State	Goal	Expression	Term	"a"	"-"	"."
1	Accept	Shift 2	Shift 3	Shift 4		
2					Shift 5	Reduce 1
3					Reduce 3	Reduce 3
4					Reduce 4	Reduce 4
5			Shift 6	Shift 4		
6					Reduce 2	Reduce 2

Given this table, a parse of the string "*a - a - a* ." would proceed as follows. Notice that the period has been introduced merely to make recognizing the end of the string somewhat easier.

State	Symbol	Stack	Action
1	*a*	1	Shift to state 4, accept *a*
4	-	1 4	Reduce by (4) *Term = a*
1	*Term*	1	Shift to state 3
3	-	1 3	Reduce by (3) *Expression = Term*
1	*Expression*	1	Shift to state 2
2	-	1 2	Shift to state 5, accept -
5	*a*	1 2 5	Shift to state 4, accept *a*
4	-	1 2 5 4	Reduce by (4) *Term = a*
5	*Term*	1 2 5	Shift to state 6
6	-	1 2 5 6	Reduce by (2) *Expression = Expression - Term*
1	*Expression*	1	Shift to state 2
2	-	1 2	Shift to state 5, accept -
5	*a*	1 2 5	Shift to state 4, accept *a*
4	.	1 2 5 4	Reduce by (4) *Term = a*
5	*Term*	1 2 5	Shift to state 6
6	.	1 2 5 6	Reduce by (2) *Expression = Expression - Term*
1	*Expression*	1	Shift to state 2
2	.	1 2	Reduce by (1) *Goal = Expression*
1	*Goal*	1	Accept as completed

The reader will have noticed that the parsing table for the toy example is very sparsely filled. The use of fixed size arrays for this, for the production lists, or for the parse stack is clearly non-optimal.

One of the great problems in using the LR method in real applications is the amount of storage which these structures require, and considerable research has been done so as to minimize this.

As in the case of LL(1) parsers it is necessary to ensure that productions are of the correct form before we can write a deterministic parser using such algorithms. Technically one has to avoid what are known as "shift/reduce conflicts", or ambiguities in the action that is needed at each entry in the parse table. In practice the difficult task of producing the parse table for a large grammar with many productions and many states, and of checking for such conflicts, is invariably left to parser generator programs, of which the best known is probably **yacc** (Johnson, 1975). A discussion of yacc, and of its underlying algorithms for LR(k) parsing is, regrettably, beyond the scope of this book.

It turns out that LR(k) parsing is much more powerful than LL(k) parsing. Before an LL(1) parser can be written it may be necessary to transform an intuitively obvious grammar into one for which the LL(1) conditions are met, and this sometimes leads to grammars that look unnaturally complicated. Fewer transformations of this sort are needed for LR(k) parsers - for example, left recursion does not present a problem, as can be seen from the simple example discussed earlier. On the other hand, when a parser is extended to handle constraint analysis and code generation, an LL(1)-based grammar presents fewer problems than does an LR(1)-based one, where the extensions are sometimes found to introduce violations of the LR(k) rules, resulting in the need to transform the grammar anyway.

The rest of our treatment will all be presented in terms of the recursive descent technique, which has the great advantage that it is intuitively easy to understand, is easy to incorporate into hand-crafted compilers, and leads to small and efficient compilers.

Further reading

On the accompanying diskette will be found source code for a demonstration program that implements the above algorithm in the case where the symbols can be represented by single characters. The reader may like to experiment with this, but be warned that the simplicity of the parsing algorithm is rather overwhelmed by all the code required to read in the productions and the elements of the parsing tables.

In the original explanation of the method we demonstrated the use of a stack which contained symbols; in the later discussion we commented that the algorithm could merely stack states. However, for demonstration purposes it is convenient to show both these structures, and so in the program we have made use of a **variant record** or **union** for handling the parse stack, so as to accommodate elements which represent symbols as well as ones which represent parse states. An alternative method would be to use two separate stacks, as is outlined by Hunter (1981).

Good discussions of LR(k) parsing and of its variations such as SLR (Simple LR) and LALR (Look Ahead LR) appear in many of the sources mentioned earlier in this chapter. (These variations aim to reduce the size of the parsing tables, at the cost of being able to handle slightly less general grammars.) The books by Gough (1988) and by Fischer and LeBlanc (1988, 1991) have useful comparisons of the relative merits of LL(k) and LR(k) parsing techniques.

10.7 Automated construction of scanners and parsers

Recursive descent parsers are easily written, provided a satisfactory grammar can be found. Since the code tends to match the grammar very closely, they may be developed manually quickly and accurately. Similarly, for many applications the manual construction of scanners using the techniques demonstrated in the last section turns out to be straightforward.

However, as with so many "real" programming projects, when one comes to develop a large compiler, the complexities of scale raise their ugly heads. An obvious course of action is to interleave the parser with the semantic analysis and code generation phases. Even when modular techniques are used - such as writing the system to encapsulate the phases in well-defined separate classes or modules - real compilers all too easily become difficult to understand, or to maintain (especially in a "portable" form).

For this reason, among others, increasing use is now made of **parser generators** and **scanner generators** - programs that take for their input a system of productions and create the corresponding parsers and scanners automatically. We have already made frequent reference to one such tool, Coco/R (Mössenböck, 1990a), which exists in a number of versions that can generate systems, embodying recursive descent parsers, in either C, C++, Java, Pascal, Modula-2 or Oberon. We shall make considerable use of this tool in the remainder of this text.

Elementary use of a tool like Coco/R is deceptively easy. The user prepares a Cocol grammar description of the language for which the scanner and parser are required. This grammar description forms the most obvious part of the input to Coco/R. Other parts come in the form of so-called **frame files** that give the skeleton of the common code that is to be generated for any scanner, parser or driver program. Such frame files are highly generic, and a user can often employ a standard set of frame files for a wide number of applications.

The tool is typically invoked with a command like

```
cocor -c -l -f grammarName
```

where `grammarName` is the name of the file containing the Cocol description. The arguments prefixed with hyphens are used in the usual way to select various options, such as the generation of a driver module (`-c`), the production of a detailed listing (`-l`), a summary of the FIRST and FOLLOW sets for each non-terminal (`-f`), and so on.

After the grammar has been analysed and tested for self-consistency and correctness (ensuring, for example, that all non-terminals have been defined, that there are no circular derivations, and that all tokens can be distinguished), a recursive descent parser and complementary FSA scanner are generated in the form of highly readable source code. The exact form of this depends on the version of Coco/R that is being used. The Modula-2 version, for example, generates `DEFINITION MODULES` specifying the interfaces, along with `IMPLEMENTATION MODULES` detailing the implementation of each component, while the C++ version produces separate header and implementation files that define a hierarchical set of classes.

Of course, such tools can only be successfully used if the user understands the premises on which they are based (for example, Coco/R can guarantee real success only if it is presented with an underlying grammar that is LL(1)). Their full power comes about when the grammar descriptions are extended further in ways to be described in the next chapter, allowing for the construction of complete compilers incorporating constraint analysis, error recovery, and code generation, and so

we delay further discussion for the present.

Exercises

10.12 On the accompanying diskette will be found implementations of Coco/R for C/C++, Turbo Pascal, and Modula-2. Submit the sample grammar given earlier to the version of your choice, and compare the code generated with that produced by hand in earlier sections.

10.13 Exercises 5.11 through 5.21 required you to produce Cocol descriptions of a number of grammars. Submit these to Coco/R and explore its capabilities for testing grammars, listing FIRST and FOLLOW sets, and constructing scanners and parsers.

Further reading

Probably the most famous parser generator is **yacc**, originally developed by Johnson (1975). There are several excellent texts that describe the use of **yacc** and its associated scanner generator **lex** (Lesk, 1975), for example those by Aho, Sethi and Ullman (1986), Bennett (1990), Levine, Mason and Brown (1992), and Schreiner and Friedman (1985).

The books by Fischer and LeBlanc (1988) and Alblas and Nymeyer (1996) describe other generators written in Pascal and in C respectively.

There are now a great many compiler generating toolkits available. Many of them are freely available from one or other of the large repositories of software on the Internet (some of these are listed in Appendix A). The most powerful are more difficult to use than Coco/R, offering, as they do, many extra features, and, in particular, incorporating more sophisticated error recovery techniques than are found in Coco/R. It will suffice to mention three of these.

Grosch (1988, 1989, 1990a), has developed a toolkit known as Cocktail, with components for generating LALR based parsers (LALR), recursive descent parsers (ELL), and scanners (REX), in a variety of languages.

Grune and Jacobs (1988) describe their LL(1)-based tool (LLGen), as a "programmer friendly LL(1) parser". It incorporates a number of interesting techniques for helping to resolve LL(1) conflicts, improving error recovery, and speeding up the development of large grammars.

A toolkit for generating compilers written in C or C++ that has received much attention is PCCTS, the Purdue University Compiler Construction Tool Set (Parr, Dietz and Cohen (1992), Parr (1996)). This is comprised of a parser generator (ANTLR), a scanner generator (DLG) and a tree-parser generator (SORCERER). It provides internal support for a number of frequently needed operations (such as abstract syntax tree construction), and is particularly interesting in that it uses LL(k) parsing with $k > 1$, which its authors claim give it a distinct edge over the more traditional LL(1) parsers (Parr and Quong, 1995, 1996).

11 SYNTAX-DIRECTED TRANSLATION

In this chapter we build on the ideas developed in the last two, and continue towards our goal of developing translators for computer languages, by discussing how syntax analysis can form the basis for driving a translator, or similar programs that process input strings that can be described by a grammar. Our discussion will be limited to methods that fit in with the top-down approach studied so far, and we shall make the further simplifying assumption that the sentences to be analysed are essentially syntactically correct.

11.1 Embedding semantic actions into syntax rules

The primary goal of the types of parser studied in the last chapter - or, indeed, of any parser - is the recognition or rejection of input strings that claim to be valid sentences of the language under consideration. However, it does not take much imagination to see that once a parser has been constructed it might be enhanced to perform specific actions whenever various syntactic constructs have been recognized.

As usual, a simple example will help to crystallize the concept. We turn again to the grammars that can describe simple algebraic expressions, and in this case to a variant that can handle parenthesized expressions in addition to the usual four operators:

```
Expression =  Term { "+" Term | "-" Term } .
Term       =  Factor { "*" Factor | "/" Factor } .
Factor     =  identifier | number | "(" Expression ")" .
```

It is easily verified that this grammar is LL(1). A simple recursive descent parser is readily constructed, with the aim of accepting a valid input expression, or aborting with an appropriate message if the input expression is malformed.

```
void Expression(void);  // function prototype

void Factor(void)
// Factor = identifier | number | "(" Expression ")" .
{ switch (SYM.sym)
  { case identifier:
    case number:
      getsym(); break;
    case lparen:
      getsym(); Expression();
      accept(rparen, " Error - ')' expected"); break;
    default:
      printf("Unexpected symbol\n"); exit(1);
  }
}

void Term(void)
// Term = Factor { "*" Factor | "/" Factor } .
{ Factor();
  while (SYM.sym == times || SYM.sym == slash)
  { getsym(); Factor(); }
}

void Expression(void)
// Expression = Term { "+" Term | "-" Term } .
{ Term();
  while (SYM.sym == plus || SYM.sym == minus)
  { getsym(); Term(); }
}
```

Note that in this and subsequent examples we have assumed the existence of a lower level scanner that recognizes fundamental terminal symbols, and constructs a globally accessible variable `SYM` that has a structure declared on the lines of

```
enum symtypes {
  unknown, eofsym, identifier, number, plus, minus, times, slash,
  lparen, rparen, equals
};

struct symbols {
  symtypes sym;     // class
  char name;        // lexeme
  int num;          // value
};

symbols SYM; // Source token
```

The parser proper requires that an initial call to `getsym()` be made before calling `Expression()` for the first time.

We have also assumed the existence of a severe error handler, similar to that used in the last chapter:

```
void accept(symtypes expectedterminal, char *errormessage)
{ if (SYM.sym != expectedterminal) { puts(errormessage); exit(1); }
  getsym();
}
```

Now consider the problem of reading a valid string in this language, and translating it into a string that has the same meaning, but which is expressed in *postfix* (that is, "reverse Polish") notation. Here the operators follow the pair-wise operands, and there is no need for parentheses. For example, the infix expression

(*a* + *b*) * (*c* - *d*)

is to be translated into its postfix equivalent

a b + *c d* - *

This is a well-known problem, admitting of a fairly straightforward solution. As we read the input string from left to right we immediately copy all the operands to the output stream as soon as they are recognized, but we delay copying the operators until we can do so in an order that relates to the familiar precedence rules for the operations. With a little thought the reader should see that the grammar and the parser given above capture the spirit of these precedence rules. Given this insight, it is not difficult to see that the augmented routine below not only parses input strings; the execution of the carefully positioned output statements effectively produces the required postfix translation.

```
void Factor(void)
// Factor = identifier | number | "(" Expression ")" .
{ switch (SYM.sym)
  { case identifier:
    case number:
      printf(" %c ", SYM.name); getsym(); break;
    case lparen:
      getsym(); Expression();
      accept(rparen, " Error - ')' expected"); break;
    default:
      printf("Unexpected symbol\n"); exit(1);
  }
}

void Term(void)
// Term = Factor { "*" Factor | "/" Factor } .
{ Factor();
  while (SYM.sym == times || SYM.sym == slash)
```

```
  { switch (SYM.sym)
    { case times: getsym(); Factor(); printf(" * "); break;
      case slash: getsym(); Factor(); printf(" / "); break;
    }
  }
}

void Expression(void)
// Expression = Term { "+" Term | "-" Term } .
{ Term();
  while (SYM.sym == plus || SYM.sym == minus)
  { switch (SYM.sym)
    { case plus:  getsym(); Term(); printf(" + "); break;
      case minus: getsym(); Term(); printf(" - "); break;
    }
  }
}
```

In a very real sense we have moved from a parser to a compiler in one easy move! What we have illustrated is a simple example of a syntax-directed program; one in which the underlying algorithm is readily developed from an understanding of an underlying syntactic structure. Compilers are obvious candidates for this sort of development, although the technique is more generally applicable, as hopefully will become clear.

The reader might wonder whether this idea could somehow be reflected back to the formal grammar from which the parser was developed. Various schemes have been proposed for doing this. Many of these use the idea of adding **semantic actions** into context-free BNF or EBNF production schemes.

Unfortunately there is no clear winner among the notations proposed for this purpose. Most, however, incorporate the actions by writing statements in some implementation language (for example, Modula-2 or C++) between suitably chosen meta-brackets that are not already bespoke in that language. For example, Coco/R uses EBNF for expressing the productions and brackets the actions with "(." and ".)", as in the example below.

```
Expression
=  Term
   {   "+" Term             (. Write('+'); .)
     | "-" Term             (. Write('-'); .)
   } .

Term
=  Factor
   {   "*" Factor           (. Write('*'); .)
     | "/" Factor           (. Write('/'); .)
   } .

Factor
=   ( identifier | number ) (. Write(SYM.name); .)
  | "("  Expression  ")" .
```

The **yacc** parser generator on UNIX systems uses unextended BNF for the productions and uses braces "{" and "}" around actions expressed in C.

Exercises

11.1 Extend the grammar and the parsers so as to handle an expression language in which one may have an optional leading + or - sign (as exemplified by + *a* * (- *b* + *c*)).

11.2 Attribute grammars

A little reflection will show that, although an algebraic expression clearly has a semantic meaning (in the sense of its "value"), this was not brought out when developing the last example. While the idea of incorporating actions into the context-free productions of a grammar gives a powerful tool for documenting and developing syntax- directed programs, what we have seen so far is still inadequate for handling the many situations where some deeper semantic meaning is required.

We have seen how a context-free grammar can be used to describe many features of programming languages. Such grammars effectively define a derivation or parse tree for each syntactically correct program in the language, and we have seen that with care we can construct the grammar so that a parse tree in some way reflects the meaning of the program as well.

As an example, consider the usual old chestnut language, albeit expressed with a slightly different (non-LL(1)) grammar

```
Goal       =  Expression .
Expression =  Term  |  Expression  "+"  Term  |  Expression  "-"  Term.
Term       =  Factor  |  Term  "*"  Factor  |  Term  "/"  Factor .
Factor     =  identifier  |  number  |  "("  Expression  ")" .
```

and consider the phrase structure tree for *x + y * z*, shown in Figure 11.1.

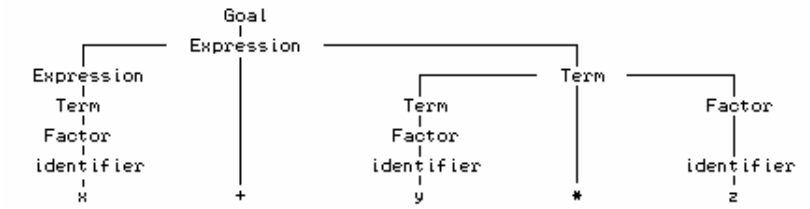


Figure 11.1 Phrase structure tree for x + y * z

Suppose *x*, *y* and *z* had associated numerical values of 3, 4 and 5, respectively. We can think of these as **semantic attributes** of the leaf nodes *x*, *y* and *z*. Similarly we can think of the nodes '+' and '*' as having attributes of "add" and "multiply". Evaluation of the whole expression can be regarded as a process where these various attributes are passed "up" the tree from the terminal nodes and are semantically transformed and combined at higher nodes to produce a final result or attribute at the root - the value (23) of the *Goal* symbol. This is illustrated in Figure 11.2.

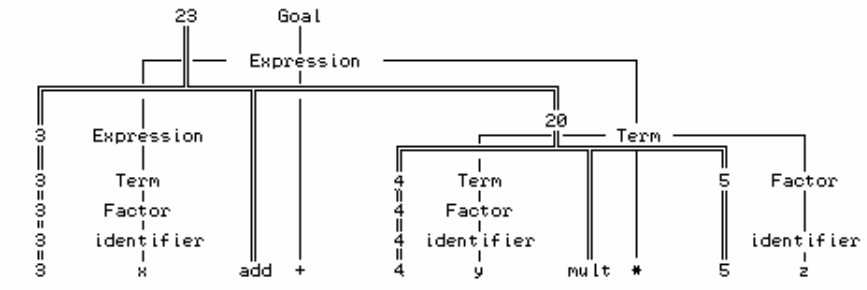


Figure 11.2 Passing attributes up a parse tree

In principle, and indeed in practice, parsing algorithms can be written whose embedded actions explicitly construct such trees as the input sentences are parsed, and also *decorate* or *annotate* the nodes with the semantic attributes. Associated tree-walking algorithms can then later be invoked to

process this semantic information in a variety of ways, possibly making several passes over the tree before the evaluation is complete. This approach lends itself well to the construction of optimizing compilers, where repeatedly walking the tree can be used to prune or graft nodes in a way that a simpler compiler cannot hope to do.

The parser constructed in the last section for recognizing this language did not, of course, construct an explicit parse tree. The grammar we have now employed seems to map immediately to parse trees in which the usual associativity and precedence of the operators is correctly reflected. It is left recursive, and thus unsuitable as the basis on which to construct a recursive descent parser. However, as we saw in section 10.6, it is possible to construct other forms of parser to handle grammars that employ left recursion. For the moment we shall not pursue the interesting problem of whether or how a recursive descent parser could be modified to generate an explicit tree. We shall content ourselves with the observation that the execution of such a parser effectively walks an implicit structure, whose nodes correspond to the various calls made to the sub-parsers as the parse proceeds.

Notwithstanding any apparent practical difficulties, our notions of formal grammars may be extended to try to capture the essence of the attributes associated with the nodes, by extending the notation still further. In one scheme, attribute rules are associated with the context-free productions in much the same way as we have already seen for actions, giving rise to what is known as an **attribute grammar**. As usual, an example will help to clarify:

```
Goal
=   Expression            (. Goal.Value := Expr.Value .) .
Expression
=   Term                  (. Expr.Value := Term.Value .)
  | Expression  "+"  Term (. Expr.Value := Expr.Value + Term.Value .)
  | Expression  "-"  Term (. Expr.Value := Expr.Value - Term.Value .) .
Term
=   Factor                (. Term.Value := Fact.Value .)
  | Term  "*"  Factor     (. Term.Value := Term.Value * Fact.Value .)
  | Term  "/"  Factor     (. Term.Value := Term.Value / Fact.Value .) .
Factor
=   identifier            (. Fact.Value := identifier.Value .)
  | number                (. Fact.Value := number.Value .)
  | "("  Expression  ")"  (. Fact.Value := Expr.Value .) .
```

Here we have employed the familiar "dot" notation that many imperative languages use in designating the elements of record structures. Were we to employ a parsing algorithm that constructed an explicit tree, this notation would immediately be consistent with the declarations of the tree nodes used for these structures.

It is important to note that the semantic rules for a given production specify the relationships between attributes of other symbols in the *same* production, and are essentially "local".

It is not necessary to have a left recursive grammar to be able to provide attribute information. We could write an iterative LL(1) grammar in much the same way:

```
Goal
=  Expression            (. Goal.Value := Expr.Value .) .
Expression
=  Term                  (. Expr.Value := Term.Value .)
   {   "+"  Term         (. Expr.Value := Expr.Value + Term.Value .)
     | "-"  Term         (. Expr.Value := Expr.Value - Term.Value .)
   } .
Term
=  Factor                (. Term.Value := Fact.Value .)
   {   "*"  Factor       (. Term.Value := Term.Value * Fact.Value .)
     | "/"  Factor       (. Term.Value := Term.Value / Fact.Value .)
   } .
Factor
=   identifier           (. Fact.Value := identifier.Value .)
  | number               (. Fact.Value := number.Value .)
```

```
    | "("  Expression  ")"  (. Fact.Value := Expr.Value .) .
```

Our notation does yet lend itself immediately to the specification and construction of those parsers that do *not* construct explicit structures of decorated nodes. However, it is not difficult to develop a suitable extension. We have already seen that the construction of parsers can be based on the idea that expansion of each non-terminal is handled by an associated routine. These routines can be parameterized, and the parameters can transmit the attributes to where they are needed. Using this idea we might express our expression grammar as follows (where we have introduced yet more meta-brackets, this time denoted by "<" and ">"):

```
Goal < Value >
=  Expression  < Value > .
Expression < Value >
=  Term < Value >
   {   "+"  Term < TermValue >     (. Value := Value + TermValue .)
     | "-"  Term < TermValue >     (. Value := Value - TermValue .)
   } .
Term < Value >
=  Factor < Value >
   {   "*"  Factor < FactorValue > (. Value := Value * FactorValue .)
     | "/"  Factor < FactorValue > (. Value := Value / FactorValue .)
   } .
Factor < Value >
=    identifier < Value >
   |  number  < Value >
   |  "("  Expression  < Value > ")" .
```

11.3 Synthesized and inherited attributes

A little contemplation of the parse tree in our earlier example, and of the attributes as given here, should convince the reader that (in this example at least) we have a situation in which the attributes of any particular node depend only on the attributes of nodes in the subtrees of the node in question. In a sense, information is always passed "up" the tree, or "out" of the corresponding routines. The parameters must be passed "by reference", and the grammar above maps into code of the form shown below (where we shall, for the moment, ignore the issue of how one attributes an identifier with an associated numeric value).

```
void Factor(int &value)
// Factor = identifier | number | "(" Expression ")" .
{ switch (SYM.sym)
  { case identifier:
    case number:
      value = SYM.num; getsym(); break;
    case lparen:
      getsym(); Expression(value);
      accept(rparen, " Error - ')' expected"); break;
    default:
      printf("Unexpected symbol\n"); exit(1);
  }
}

void Term(int &value)
// Term = Factor { "*" Factor | "/" Factor } .
{ int factorvalue;
  Factor(value);
  while (SYM.sym == times || SYM.sym == slash)
  { switch (SYM.sym)
    { case times:
        getsym(); Factor(factorvalue); value *= factorvalue; break;
      case slash:
        getsym(); Factor(factorvalue); value /= factorvalue; break;
    }
  }
}

void Expression(int &value)
// Expression = Term { "+" Term | "-" Term } .
{ int termvalue;
```

```
    Term(value);
    while (SYM.sym == plus || SYM.sym == minus)
    { switch (SYM.sym)
      { case plus:
          getsym(); Term(termvalue); value += termvalue; break;
        case minus:
          getsym(); Term(termvalue); value -= termvalue; break;
      }
    }
  }
```

Attributes that travel in this way are known as synthesized attributes. In general, given a context-free production rule of the form

$$A = \alpha\ B\ \gamma$$

then an associated semantic rule of the form

$$A.\text{attribute}_i = f\ (\alpha.\text{attribute}_j, B.\text{attribute}_k, \gamma.\text{attribute}_l)$$

is said to specify a **synthesized attribute** of *A*.

Attributes do not always travel up a tree. As a rather grander example, consider the very small CLANG program:

```
PROGRAM Silly;
  CONST
    Bonus = 4;
  VAR
    Pay;
  BEGIN
    WRITE(Pay + Bonus)
  END.
```

which has the phrase structure tree shown in Figure 11.3.

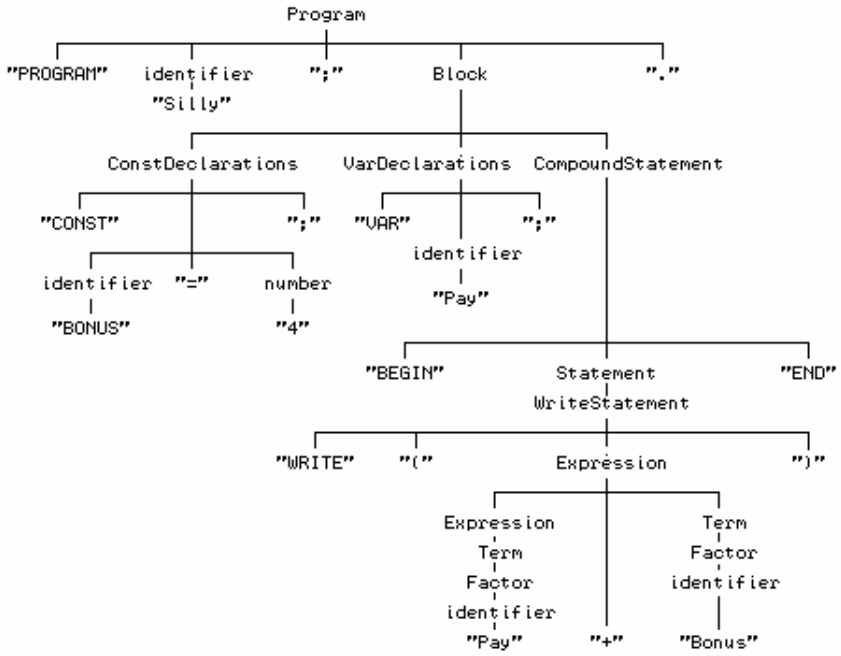


Figure 11.3 Parse tree for a complete small program

In this case we can think of the Boolean *IsConstant* and *IsVariable* attributes of the nodes `CONST` and `VAR` as being passed up the tree (*synthesized*), and then later passed back down and *inherited* by

other nodes like `Bonus` and `Pay` (see Figure 11.4). In a sense, the context in which the identifiers were declared is being remembered - the system is providing a way of handling context-sensitive features of an otherwise context-free language.

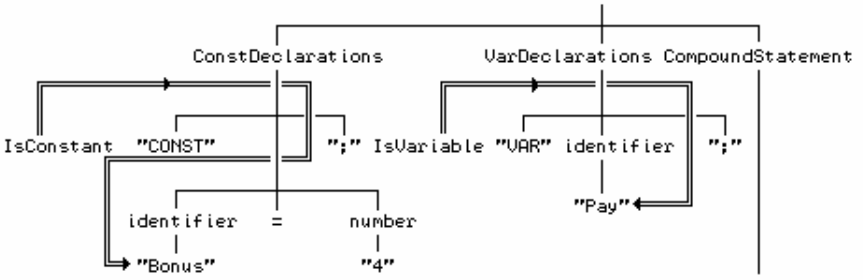


Figure 11.4 Attributes passed up and down a parse tree

Of course, this idea must be taken much further. Attributes like this form part of what is usually termed an **environment**. Compilation or parsing of programs in a language like Pascal or Modula-2 generally begins in a "standard" environment, into which pervasive identifiers like `TRUE`, `FALSE`, `ORD`, `CHR` and so on are already incorporated. This environment is inherited by *Program* and then by *Block* and then by *ConstDeclarations*, which augments it and passes it back up, to be inherited in its augmented form by *VarDeclarations* which augments it further and passes it back, so that it may then be passed down to the *CompoundStatement*. We may try to depict this as shown in Figure 11.5.

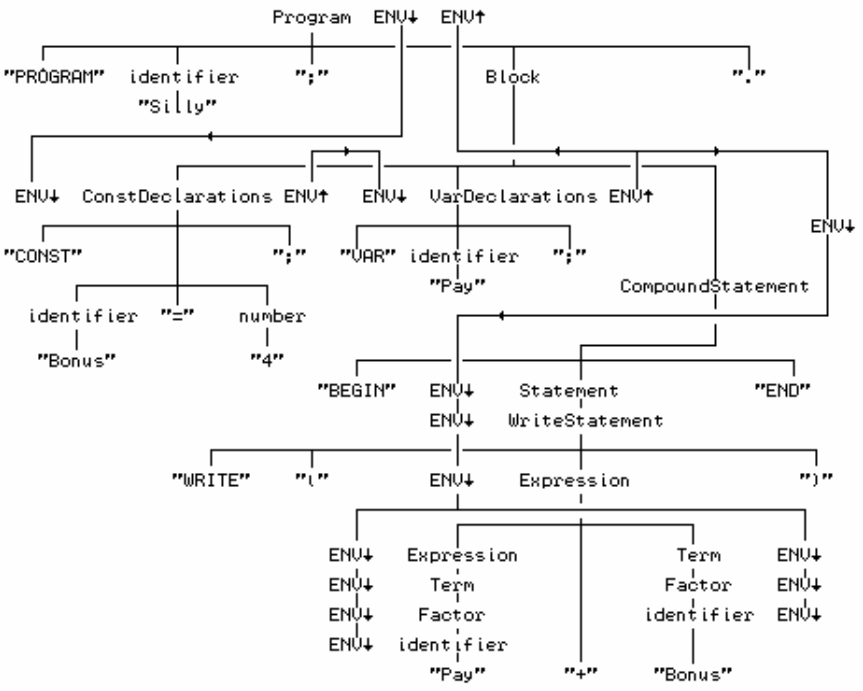


Figure 11.5 Modification of the parsing environment

More generally, given a context-free production rule of the form

$$A = \alpha\ B\ \gamma$$

an associated semantic rule of the form

$$B.\text{attribute}_i = f\,(\alpha.\text{attribute}_j, A.\text{attribute}_k, \gamma.\text{attribute}_l)$$

is said to specify an **inherited attribute** of *B*. The inherited attributes of a symbol are computed from information held in the environment of the symbol in the parse tree.

As before, our formal notation needs modification to reflect the different forms and flows of attributes. A notation often used employs arrows ↑ and ↓ in conjunction with the parameters mentioned in the < > metabrackets. Inherited attributes are marked with ↓, and synthesized attributes with ↑. In terms of actual coding, ↑ attributes correspond to "reference" parameters, while ↓ attributes correspond to "value" parameters. In practice, reference parameters may also be used to manipulate features (such as an environment) that are inherited, modified, and then returned; these are sometimes called **transmitted attributes**, and are marked with ↓↑ or ↕.

11.4 Classes of attribute grammars

Attribute grammars have other important features. If the action of a parser is in some way to construct a tree whose nodes are decorated with semantic attributes relevant to each node, then "walking" this tree after it has been constructed should constitute a possible mechanism for developing the synthetic aspects of a translator, such as code generation. If this is this case, then the order in which the tree is walked becomes crucially important, since attributes can depend on one another. The simplest tree walk - the depth-first, left-to-right method - may not suffice. Indeed, we have a situation completely analogous to that which arises in attempting single-pass assembly and discovering forward references to labels. In principle we can, of course, perform multiple tree walks, just as we can perform multiple-pass assembly. There are, however, two types of attribute grammars for which this is not necessary.

- An **S-attributed grammar** is one that uses only synthesized attributes. For such a grammar the attributes can obviously be correctly evaluated using a bottom-up walk of the parse tree. Furthermore, such a grammar is easily handled by parsing algorithms (such as recursive descent) that do not explicitly build the parse tree.

- An **L-attributed grammar** is one in which the inherited attributes of a particular symbol in any given production are restricted in certain ways. For each production of the general form

  $$A \rightarrow B_1\ B_2\ \ldots\ B_n$$

  the inherited attributes of B_k may depend only on the inherited attributes of *A* or synthesized attributes of $B_1, B_2 \ldots B_{k-1}$. For such a grammar the attributes can be correctly evaluated using a left- to-right depth-first walk of the parse tree, and such grammars are usually easily handled by recursive descent parsers, which implicitly walk the parse tree in this way.

We have already pointed out that there are various aspects of computer languages that involve context sensitivity, even though the general form of the syntax might be expressed in a context-free way. Context-sensitive constraints on such languages - often called *context conditions* - are often conveniently expressed by conditions included in its attribute grammar, specifying relations that must be satisfied between the attribute values in the parse tree of a valid program. For example, we might have a production like

```
Assignment =  VarDesignator < TypeV↑ > ":=" Expression < TypeE↑ >
              (. where AssignmentCompatible(TypeV↓, TypeE↓) .) .
```

Alternatively, and more usefully in the construction of real parsers, the context conditions might be expressed in the same notation as for semantic actions, for example

```
Assignment =  VarDesignator < TypeV↑ > ":=" Expression < TypeE↑ >
              (. if (Incompatible(TypeV↓, TypeE↓))
                     SemanticError("Incompatible types"); .) .
```

Finally, we should note that the concept of an attribute grammar may be formally defined in several ways. Waite and Goos (1984) and Rechenberg and Mössenböck (1989) suggest:

> An attribute grammar is a quadruple { *G, A, R, K* }, where *G* = { *N, T, S, P* } is a reduced context-free grammar, *A* is a finite set of attributes, *R* is a finite set of semantic actions, and *K* is a finite set of context conditions. Zero or more attributes from *A* are associated with each symbol $X \in N \cup T$, and zero or more semantic actions from *R* and zero or more context conditions from *K* are associated with each production in *P*. For each occurrence of a non-terminal *X* in the parse tree of a sentence in *L*(*G*) the attributes of *X* can be computed in at most one way by semantic actions.

Further reading

Good treatments of the material discussed in this section can be found in the books by Gough (1988), Bennett (1990), and Rechenberg and Mössenböck (1989). As always, the text by Aho, Sethi and Ullman (1986) is a mine of information.

11.5 Case study - a small student database

As another example of using an attribute grammar to construct a system, consider the problem of constructing a database of the members of a student group. In particular, we wish to record their names, along with their intended degrees, after extracting information from an original data file that has records like the following:

```
CompScience3
    BSc  : Mike, Juanito, Rob, Keith, Bruce ;
    BScS : Erik, Arne, Paul, Rory, Andrew, Carl, Jeffrey ;
    BSc  : Nico, Kirsten, Peter, Luanne, Jackie, Mark .
```

Although we are not involved with constructing a compiler in this instance, we still have an example of a syntax directed computation. This data can be described by the context-free productions

```
ClassList  =  ClassName [ Group { ";" Group  } ]  "." .
Group      =  Degree  ":"  Student { "," Student } .
Degree     =  "BSc" | "BScS" .
ClassName  =  identifier .
Student    =  identifier .
```

The attributes of greatest interest are, probably, those that relate to the students' names and degree codes. An attribute grammar, with semantic actions that define how the database could be set up, is as follows:

```
ClassList
=  ClassName                (. OpenDataBase .)
   [ Group { ";" Group } ]  (. CloseDataBase .)
   "." .
Group
=  Degree < DegreeCode↑ >
   ":"  Student < DegreeCode↓ >
   { "," Student < DegreeCode↓ > } .
Degree < DegreeCode ↑ >
=   "BSc"                   (. DegreeCode := bsc .)
  | "BScS"                  (. DegreeCode := bscs .) .
ClassName
=  identifier .
Student < DegreeCode↓ >
=  identifier < Name↑ >     (. AddToDataBase(Name↓, DegreeCode↓) .) .
```

It should be easy to see that this can be used to derive code on the lines of

```
void Student(codes DegreeCode)
{ if (SYM.sym == identifier)
    { AddToDataBase(SYM.name, DegreeCode); getsym(); }
  else
    { printf(" error - student name expected\n"); exit(1); }
}

void Degree(codes &DegreeCode)
{ switch (SYM.sym)
  { case bscsym  : DegreeCode = bsc; break;
    case bscssym : DegreeCode = bscs; break;
    default : printf(" error - invalid degree\n"); exit(1);
  }
  getsym();
}

void Group(void)
{ codes DegreeCode;
  Degree(DegreeCode);
  accept(colon, " error - ':' expected");
  Student(DegreeCode);
  while (SYM.sym == comma)
  { getsym(); Student(DegreeCode); }
}

void ClassName(void)
{ accept(identifier, " error - class name expected"); }

void ClassList(void)
{ ClassName();
  OpenDataBase();
  if (SYM.sym == bscsym || SYM.sym == bscssym)
  { Group();
    while (SYM.sym == semicolon) { getsym(); Group(); }
  }
  CloseDataBase();
  accept(period, " error - '.' expected");
}
```

Although all the examples so far have lent themselves to very easy implementation by a recursive descent parser, it is not difficult to find an example where difficulties arise. Consider the *ClassList* example again, but suppose that the input data had been of a form like

```
CompScience3
   Mike, Juanito, Rob, Keith, Bruce              : BSc ;
   Erik, Arne, Paul, Rory, Andrew, Carl, Jeffrey : BScS ;
   Nico, Kirsten, Peter, Luanne, Jackie, Mark    : BSc .
```

This data can be described by the context-free productions

```
ClassList  =  ClassName [ Group { ";" Group } ]  "." .
Group      =  Student { "," Student } ":" Degree .
Degree     =  "BSc" | "BScS" .
ClassName  =  identifier .
Student    =  identifier .
```

Now a moment's thought should convince the reader that attributing the grammar as follows

```
Group
=  Student < Name↑ >           (. AddToDataBase(Name↓, DegreeCode↓) .)
   { "," Student < Name↑ >     (. AddToDataBase(Name↓, DegreeCode↓) .)
   } ":" Degree < DegreeCode↑ > .
Student < Name↑ >
=  identifier < Name↑ >
```

does not create an L-attributed grammar, but has the unfortunate effect that at the point where it seems natural to add a student to the database, his or her degree has not yet been ascertained.

Just as we did for the one-pass assembler, so here we can sidestep the problem by creating a local forward reference table. It is not particularly difficult to handle this grammar with a recursive descent parser, as the following amended code will reveal:

```
void Student(names &Name)
{ if (SYM.sym == identifier)
    { Name = SYM.name; getsym(); }
  else
    { printf(" error - student name expected\n"); exit(1); }
}

void Group(void)
{ codes DegreeCode;
  names Name[100];
  int last = 0;
  Student(Name[last]);            // first forward reference
  while (SYM.sym == comma)
  { getsym();
    last++; Student(Name[last]);  // add to forward references
  }
  accept(colon, " error - ':' expected");
  Degree(DegreeCode);
  for (int i = last; i >= 0; i--) // process forward reference list
    AddToDataBase(Name[i], DegreeCode);
}
```

Exercises

11.2 Develop an attribute grammar and corresponding parser to handle the evaluation of an expression where there may be an optional leading `+` or `-` sign (as exemplified by `+ 9 * ( - 6 + 5 )`).

11.3 Develop an attribute grammar for the 8-bit ASSEMBLER language used in section 4.3, and use it to build an assembler for this language.

11.4 Develop an attribute grammar for the stack ASSEMBLER language used in section 4.4, and use it to build an assembler for this language.

12 USING COCO/R - OVERVIEW

One of the main reasons for developing attributed grammars like those discussed in the last chapter is to be able to use them as input to compiler generator tools, and so construct complete programs. It is the aim of this chapter and the next to illustrate how this process is achieved with Coco/R, and to discuss the Cocol specification language in greater detail than before. Our discussion will, as usual, focus mainly on C++ applications, but a study of the documentation and examples on the diskette should allow Modula-2, Pascal and "traditional C" readers to develop in those languages just as easily.

12.1 Installing and running Coco/R

On the diskette that accompanies this book can be found three implementations of Coco/R that can generate applications in C/C++, Modula-2, or Turbo Pascal. These have been configured for easy use on MS-DOS based systems. Versions of Coco/R are also available for use with many other compilers and operating systems. These can be obtained from several sites on the Internet; a list of some of these appears in Appendix A.

The installation and execution of Coco/R is rather system-specific, and readers will be obliged to make use of the documentation that is provided on the diskette. Nevertheless, a brief overview of the process can usefully be given here.

12.1.1 Installation

The MS-DOS versions of Coco/R are supplied as compressed, self-extracting executable files, and for these the installation process requires a user to

- create a system directory to store the system files [`MKDIR C:\COCO`];
- make this the active directory [`CD C:\COCO`];
- copy the distribution file to the system directory [`COPY A:COCORC.EXE C:\COCO`];
- start the decompression process [`COCORC`] (this process will extract the files, and create further subdirectories to contain Coco/R and its support files and library modules);
- add the system directory to the MS-DOS "path" (this may often most easily be done by modifying the `PATH` statement in the `AUTOEXEC.BAT` file);
- compile the library support modules;
- modify the host compiler and linker parameters, so that applications created by Coco/R can easily be linked to the support modules;
- set an "environment variable", so that Coco/R can locate its "frame files" (this may often most easily be done by adding a line like `SET CRFRAMES = C:\COCO\FRAMES` to the `AUTOEXEC.BAT` file).

12.1.2 Input file preparation

For each application, the user has to prepare a text file to contain the attributed grammar. Points to be aware of are that

- it is sensible to work within a "project directory" (say `C:\WORK`) and not within the "system directory" (`C:\COCO`);
- text file preparation must be done with an ASCII editor, and not with a word processor;
- by convention the file is named with a primary name that is based on the grammar's goal symbol, and with an "`ATG`" extension, for example `CALC.ATG`.

Besides the grammar, Coco/R needs to be able to read **frame files**. These contain outlines of the scanner, parser, and driver files, to which will be added statements derived from an analysis of the attributed grammar. Frame files for the scanner and parser are of a highly standard form; the ones supplied with the distribution are suitable for use in many applications without the need for any customization. However, a complete compiler consists of more than just a scanner and parser - in particular it requires a driver program to call the parser. A basic driver frame file (`COMPILER.FRM`) comes with the kit. This will allow simple applications to be generated immediately, but it is usually necessary to copy this basic file to the project directory, and then to edit it to suit the application. The resulting file should be given the same primary name as the grammar file, and a `FRM` extension, for example `CALC.FRM`.

12.1.3 Execution

Once the input files have been prepared, generation of the application is started with a command like

```
COCOR CALC.ATG
```

A number of compiler options may be specified in a way that is probably familiar, for example

```
COCOR -L -C CALC.ATG
```

The options depend on the particular version of Coco/R in use. A summary of those available may be obtained by issuing the `COCOR` command with no parameters at all, or with only a `-H` parameter. Compiler options may also be selected by **pragmas** embedded in the attributed grammar itself, and this is probably the preferred approach for serious applications. Examples of such pragmas can be found in the case studies later in this chapter.

12.1.4 Output from Coco/R

Assuming that the attributed grammar appears to be satisfactory, and depending on the compiler switches specified, execution of Coco/R will typically result in the production of header and implementation files (with names derived from the goal symbol name) for

- a FSA scanner (for example `CALCS.HPP` and `CALCS.CPP`)
- a recursive descent parser (for example `CALCP.HPP` and `CALCP.CPP`)
- a driver routine (for example `CALC.CPP`)
- a list of error messages (for example `CALCE.H`)
- a file relating the names of tokens to the integer numbers by which they will be known to the parser (for example `CALCC.H`)

12.1.5 Assembling the generated system

After they have been generated, the various parts of an application can be compiled and linked with one another, and with any other components that they need. The way in which this is done depends very much on the host compiler. For a very simple MS-DOS application using the Borland C++ system, one might be able to use commands like

```
BCC -ml -IC:\COCO\CPLUS2 -c CALC.CPP CALCS.CPP CALCP.CPP
BCC -ml -LC:\COCO\CPLUS2 -eCALC.EXE CALC.OBJ CALCS.OBJ CALCP.OBJ CR_LIB.LIB
```

but for larger applications the use of a **makefile** is probably to be preferred. Examples of makefiles are found on the distribution diskette.

12.2 Case study - a simple adding machine

Preparation of completely attributed grammars suitable as input to Coco/R requires an in-depth understanding of the Cocol specification language, including many features that we have not yet encountered. Sections 12.3 and 12.4 discuss these aspects in some detail, and owe much to the original description by Mössenböck (1990a).

The discussion will be clarified by reference to a simple example, chosen to illustrate as many features as possible (as a result, it may appear rather contrived). Suppose we wish to construct an adding machine that can add numbers arranged in various groups into subtotals, and then either add these subtotals to a running grand total, or reject them. Our numbers can have fractional parts; just to be perverse we shall allow a shorthand notation for handling ranges of numbers. Typical input is exemplified by

```
clear                       // start the machine
10 + 20 + 3 .. 7 accept     // one subtotal 10+20+3+4+5+6+7, accepted
3.4 + 6.875..50 cancel      // another one, but rejected
3 + 4 + 6 accept            // and a third, this time accepted
total                       // display grand total and then stop
```

Correct input of this form can be described by a simple LL(1) grammar that we might try initially to specify in Cocol on the lines of the following:

```
COMPILER Calc

  CHARACTERS
    digit    = "0123456789" .

  TOKENS
    number   = digit { digit } [ "." digit { digit } ] .

  PRODUCTIONS
    Calc     = "clear" { Subtotal } "total" .
    Subtotal = Range { "+" Range } ( "accept" | "cancel" ) .
    Range    = Amount [ ".." Amount ] .
    Amount   = number .

END Calc.
```

In general a grammar like this can itself be described in EBNF by

```
Cocol =   "COMPILER" GoalIdentifier
              ArbitraryText
              ScannerSpecification
              ParserSpecification
          "END" GoalIdentifier "." .
```

We note immediately that the identifier after the keyword `COMPILER` gives the grammar name, and must match the name after the keyword `END`. The grammar name must also match the name chosen for the non-terminal that defines the goal symbol of the phrase structure grammar.

Each of the productions leads to the generation of a corresponding parsing routine. It should not take much imagination to see that the routines in our case study will also need to perform

operations like

- converting the string that defines a `number` token into a corresponding numerical value. Thus we need mechanisms for extracting attributes of the various tokens from the scanner that recognizes them.
- adding such numbers into variables declared for the purpose of recording totals and subtotals, and passing these values between the routines. Thus we need mechanisms for declaring parameters and local variables in the generated routines, and for incorporating arithmetic statements.
- displaying the values of the variables on an output device. Thus we need mechanisms for interfacing our parsing routines to external library routines.
- reacting sensibly to input data that does not conform to the proper syntax. Thus we need mechanisms for specifying how error recovery should be accomplished.
- reacting sensibly to data that is syntactically correct, but still meaningless, as might happen if one was asked to process numbers in the range 6 .. 2. Thus we need mechanisms for reporting semantic and constraint violations.

These mechanisms are all incorporated into the grammar by attributing it with extra information, as discussed in the next sections. As an immediate example of this, arbitrary text may follow the *GoalIdentifier*, preceding the *ScannerSpecification*. This is not checked by Coco/R, but is simply incorporated directly in the generated parser. This offers the facility of providing code for `IMPORT` clauses in Modula-2, `USES` clauses in Turbo Pascal, or `#include` directives in C++, and for the declaration of global objects (constants, types, variables or functions) that may be needed by later semantic actions.

12.3 Scanner specification

A scanner has to read source text, skip meaningless characters, and recognize tokens that can be handled by the parser. Clearly there has to be some way for the parser to retrieve information about these tokens. The most fundamental information can be returned in the form of a simple integer, unique to the type of token recognized. While a moment's thought will confirm that the members of such an enumeration will allow a parser to perform syntactic analysis, semantic properties (such as the numeric value of the `number` that appears in our example grammar) may require a token to be analysed in more detail. To this end, the generated scanner allows the parser to retrieve the **lexeme** or textual representation of a token.

Tokens may be classified either as literals or as token classes. As we have already seen, literals (like "`END`" and "`!=`") may be introduced directly into productions as strings, and do not need to be named. Token classes (such as *identifiers* or *numbers*) must be named, and have structures that are specified by regular expressions, defined in EBNF.

In Cocol, a scanner specification consists of six optional parts, that may, in fact, be introduced in arbitrary order.

```
ScannerSpecification =  {  CharacterSets
                         | Ignorable
                         | Comments
                         | Tokens
                         | Pragmas
                         | UserNames
                        } .
```

12.3.1 Character sets

The *CharacterSets* component allows for the declaration of names for character sets like letters or digits, and defines the characters that may occur as members of these sets. These names may then be used in the other sections of the scanner specification (but not, it should be noted, in the parser specification).

```
CharacterSets   =  "CHARACTERS" { NamedCharSet } .
NamedCharSet    =  SetIdent "=" CharacterSet "." .
CharacterSet    =  SimpleSet { ( "+" | "-" ) SimpleSet } .
SimpleSet       =  SetIdent  |  string | SingleChar [ ".." SingleChar ] | "ANY" .
SingleChar      =  "CHR" "(" number ")" .
SetIdent        =  identifier .
```

Simple character sets are denoted by one of

SetIdent	a previously declared character set with that name
String	a set consisting of all characters in the string
`CHR(`*i*`)`	a set of one character with ordinal value *i*
`CHR(`*i*`) .. CHR(`*j*`)`	a set consisting of all characters whose ordinal values are in the range *i* ... *j*.
`ANY`	the set of all characters acceptable to the implementation

Simple sets may then be combined by the union (+) and difference operators (-).

As examples we might have

```
digit     = "0123456789" .      /* The set of all digits */
hexdigit  = digit + "ABCDEF" .  /* The set of all hexadecimal digits */
eol       = CHR(10) .           /* Line feed character */
noDigit   = ANY - digit .       /* Any character that is not a digit */
ctrlChars = CHR(1) .. CHR(31) . /* The ASCII control characters */
InString  = ANY - '"' - eol .   /* Strings may not cross line boundaries */
```

12.3.2 Comments and ignorable characters

Usually spaces within the source text of a program are irrelevant, and in scanning for the start of a token, a Coco/R generated scanner will simply ignore them. Other separators like tabs, line ends, and form feeds may also be declared irrelevant, and some applications may prefer to ignore the distinction between upper and lower case input.

Comments are difficult to specify with the regular expressions used to denote tokens - indeed, nested comments may not be specified at all in this way. Since comments are usually discarded by a parsing process, and may typically appear in arbitrary places in source code, it makes sense to have a special construct to express their structure.

Ignorable aspects of the scanning process are defined in Cocol by

```
Comments  = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr [ "NESTED" ] .
Ignorable = "IGNORE" ( "CASE" | CharacterSet ) .
```

where the optional keyword `NESTED` should have an obvious meaning. A practical restriction is that comment brackets must not be longer than 2 characters. It is possible to declare several kinds of comments within a single grammar, for example, for C++:

```
COMMENTS FROM "/*" TO "*/"
COMMENTS FROM "//" TO eol
IGNORE CHR(9) .. CHR(13)
```

The set of ignorable characters in this example is that which includes the standard white space separators in ASCII files. The null character `CHR(0)` should not be included in any ignorable set. It is used internally by Coco/R to mark the end of the input file.

12.3.3 Tokens

A very important part of the scanner specification declares the form of terminal tokens:

```
Tokens       =  "TOKENS" { Token } .
Token        =  TokenSymbol [ "=" TokenExpr "." ] .
TokenExpr    =  TokenTerm { "|" TokenTerm } .
TokenTerm    =  TokenFactor { TokenFactor } [ "CONTEXT" "(" TokenExpr ")" ] .
TokenFactor  =  SetIdent | string
                  | "(" TokenExpr ")"
                  | "[" TokenExpr "]"
                  | "{" TokenExpr "}" .
TokenSymbol  =  TokenIdent | string .
TokenIdent   =  identifier .
```

Tokens may be declared in any order. A token declaration defines a *TokenSymbol* together with its structure. Usually the symbol on the left-hand side of the declaration is an identifier, which is then used in other parts of the grammar to denote the structure described on the right-hand side of the declaration by a regular expression (expressed in EBNF). This expression may contain literals denoting themselves (for example "`END`"), or the names of character sets (for example *letter*), denoting an arbitrary character from such sets. The restriction to regular expressions means that it may not contain the names of any other tokens.

While token specification is usually straightforward, there are a number of subtleties that may need emphasizing:

- Since spaces are deemed to be irrelevant when they come *between* tokens in the input for most languages, one should not attempt to declare literal tokens that have spaces *within* them.

- Our case study has introduced but one explicit token class:

  ```
  number = digit { digit } [ "." digit { digit } ] .
  ```

  However it has also introduced tokens like "`clear`", "`cancel`" and "..". This last one is particularly interesting. A scanner might have trouble distinguishing the tokens in input like

  ```
  3 .. 5.4  +  5.4..16.4  + 50..80
  ```

  because in some cases the periods form part of a real literal, in others they form part of an ellipsis. This sort of situation arises quite frequently, and Cocol makes special provision for it. An optional `CONTEXT` phrase in a *TokenTerm* specifies that this term only be recognized when its right-hand context in the input stream is the *TokenExpr* specified in brackets. Our case study example requires alteration:

  ```
  TOKENS
    number =   digit { digit } [ "." digit { digit } ]
             | digit { digit } CONTEXT ( ".." ) .
  ```

- The grammar for tokens allows for empty right-hand sides. This may seem strange, especially as no scanner is generated if the right-hand side of a declaration is missing. This facility is used if the user wishes to supply a hand-crafted scanner, rather than the one generated by Coco/R. In this case, the symbol on the left- hand side of a token declaration may also simply be specified by a *string*, with no right-hand side.

- Tokens specified without right-hand sides are numbered consecutively starting from 0, and the hand-crafted scanner has to return token codes according to this numbering scheme.

12.3.4 Pragmas

A pragma, like a comment, is a token that may occur anywhere in the input stream, but, unlike a comment, it cannot be ignored. Pragmas are often used to allow programmers to select compiler switches dynamically. Since it becomes impractical to modify the phrase structure grammar to handle this, a special mechanism is provided for the recognition and treatment of pragmas. In Cocol they are declared like tokens, but may have an associated semantic action that is executed whenever they are recognized by the scanner.

```
Pragmas     =  "PRAGMAS" { Pragma } .
Pragma      =  Token [ Action ] .
Action      =  "(." arbitraryText ".)" .
```

As an example, we might add to our case study

```
PRAGMAS
  page = "page" .  (. printf("\f"); .)
```

to allow the word `page` to appear anywhere in the input data; each appearance would have the effect of moving to a new page on the output.

12.3.5 User names

The scanner and parser produced by Coco/R use small integer values to distinguish tokens. This makes their code harder to understand by a human reader (some would argue that humans should never need to read such code anyway). When used with appropriate options, Coco/R can generate code that uses names for the tokens. By default these names have a rather stereotyped form (for example "..." would be named "`pointpointpointSym`"). The *UserNames* section may be used to prefer user-defined names, or to help resolve name clashes (for example, between the default names that would be chosen for "`point`" and ".").

```
UserNames  = "NAMES" { UserName } .
UserName   = TokenIdent  "=" ( identifier | string ) "." .
```

As examples we might have

```
NAMES
  period   = "." .
  ellipsis = "..." .
```

12.3.6 The scanner interface

The scanner generated by Coco/R declares various procedures and functions that may be called from the parser whenever it needs to obtain a new token, or to analyse one that has already been recognized. As it happens, a user rarely has to make direct use of this interface, as the generated parser incorporates all the necessary calls to the scanner routines automatically, and also provides facilities for retrieving lexemes.

The form of the interface depends on the host system. For example, for the C++ version, the interface is effectively that shown below, although there is actually an underlying class hierarchy, so that the declarations are not exactly the same as those shown. The reader should take note that there are various ways in which source text may be retrieved from the scanner (to understand these in full it will be necessary to study the class hierarchy, but easier interfaces are provided for the

parser; see section 12.4.6).

```
class grammarScanner
{ public:
    grammarScanner(int SourceFile, int ignoreCase);
    // Constructs scanner for grammar and associates this with a
    // previously opened SourceFile.  Specifies whether to IGNORE CASE

    int Get();
    // Retrieves next token from source

    void GetString(Token *Sym, char *Buffer, int Max);
    // Retrieves at most Max characters from Sym into Buffer

    void GetName(Token *Sym, char *Buffer, int Max);
    // Retrieves at most Max characters from Sym into Buffer
    // Buffer is capitalized if IGNORE CASE was specified

    long GetLine(long Pos, char *Line, int Max);
    // Retrieves at most Max characters (or until next line break)
    // from position Pos in source file into Line

};
```

12.4 Parser specification

The parser specification is the main part of the input to Coco/R. It contains the productions of an attributed grammar specifying the syntax of the language to be recognized, as well as the action to be taken as each phrase or token is recognized.

12.4.1 Productions

The form of the parser specification may itself be described in EBNF as follows. For the Modula-2 and Pascal versions we have:

```
ParserSpecification =  "PRODUCTIONS" { Production } .
Production          =  NonTerminal [ FormalAttributes ]
                         [ LocalDeclarations ]        (* Modula-2 and Pascal *)
                         "=" Expression "." .
FormalAttributes    =  "<"  arbitraryText ">" | "<."  arbitraryText ".>" .
LocalDeclarations   =  "(." arbitraryText ".)" .
NonTerminal         =  identifier .
```

For the C and C++ versions the *LocalDeclarations* follow the "=" instead:

```
Production          =  NonTerminal [ FormalAttributes ]
                         "=" [ LocalDeclarations ]  /* C and C++ */
                         Expression "." .
```

Any identifier appearing in a production that was not previously declared as a terminal token is considered to be the name of a *NonTerminal*, and there must be exactly one production for each *NonTerminal* that is used in the specification (this may, of course, specify a list of alternative right sides).

A production may be considered as a specification for creating a routine that parses the *NonTerminal*. This routine will constitute its own scope for parameters and other local components like variables and constants. The left-hand side of a *Production* specifies the name of the *NonTerminal* as well as its *FormalAttributes* (which effectively specify the formal parameters of the routine). In the Modula-2 and Pascal versions the optional *LocalDeclarations* allow the declaration of local components to precede the block of statements that follow. The C and C++ versions define their local components within this statement block, as required by the host language.

As in the case of tokens, some subtleties in the specification of productions should be emphasized:

- The productions may be given in any order.
- A production must be given for a *GoalIdentifier* that matches the name used for the grammar.
- The formal attributes enclosed in angle brackets "<" and ">" (or "<." and ".>") simply consist of parameter declarations in the host language. Similarly, where they are required and permitted, local declarations take the form of host language declarations enclosed in "(." and ".)" brackets. However, the syntax of these components is not checked by Coco/R; this is left to the responsibility of the compiler that will actually compile the generated application.
- All routines give rise to "regular procedures" (in Modula-2 terminology) or "void functions" (in C++ terminology). Coco/R cannot construct true functions that can be called from within other expressions; any return values must be transmitted using reference parameter mechanisms.
- The goal symbol may not have any *FormalAttributes*. Any information that the parser is required to pass back to the calling driver program must be handled in other ways. At times this may prove slightly awkward.
- While a production constitutes a scope for its formal attributes and its locally declared objects, terminals and non-terminals, globally declared objects, and imported modules are visible in any production.
- It may happen that an identifier chosen as the name of a *NonTerminal* may clash with one of the internal names used in the rest of the system. Such clashes will only become apparent when the application is compiled and linked, and may require the user to redefine the grammar to use other identifiers.

The *Expression* on the right-hand-side of each *Production* defines the context-free structure of some part of the source language, together with the attributes and semantic actions that specify how the parser must react to the recognition of each component. The syntax of an *Expression* may itself be described in EBNF (albeit not in LL(1) form) as

```
Expression   =  Term { "|" Term } .
Term         =  Factor { Factor }  .
Factor       =     [ "WEAK" ] TokenSymbol
                |  NonTerminal [ Attributes ]
                |  Action
                |  "ANY"
                |  "SYNC"
                |  "(" Expression ")"
                |  "[" Expression "]"
                |  "{" Expression "}" .
Attributes   =  "<"  arbitraryText ">" |  "<."  arbitraryText ".>" .
Action       =  "(." arbitraryText ".)" .
```

The *Attributes* enclosed in angle brackets that may follow a *NonTerminal* effectively denote the actual parameters that will be used in calling the corresponding routine. If a *NonTerminal* is defined on the left-hand side of a *Production* to have *FormalAttributes*, then every occurrence of that *NonTerminal* in a right-hand side *Expression* must have a list of actual attributes that correspond to the *FormalAttributes* according to the parameter compatibility rules of the host language. However, the conformance is only checked when the generated parser is itself compiled.

An *Action* is an arbitrary sequence of host language statements enclosed in "(." and ".)". These

are simply incorporated into the generated parser *in situ*; once again, no syntax is checked at that stage.

These points may be made clearer by considering a development of part of our case study, which hopefully needs little further explanation:

```
PRODUCTIONS
  Calc                                                    /* goal */
  =                 (. double total = 0.0, sub; .)        /* locals */
    "clear"
    { Subtotal<sub> (. total += sub; .)                   /* add to total */
    }
    "total"         (. printf("   total: %5.2f\n", total); .) /* display */
    .

  Subtotal<double &s>                                     /* ref param */
  =                 (. double r; .)                       /* local */
    Range<s>
    { "+" Range<r>  (. s += r; .)                         /* add to s */
    }
    (   "accept"    (. printf("subtotal: %5.2f\n", s); .) /* display */
      | "cancel"    (. s = 0.0; .)                        /* nullify */
    ) .
```

Although the input to Coco/R is free-format, it is suggested that the regular EBNF appear on the left, with the actions on the right, as in the example above.

Many aspects of parser specification are straightforward, but there are some subtleties that call for comment:

- Where it appears, the keyword `ANY` denotes any terminal that cannot follow `ANY` in that context. It can conveniently be used to parse structures that contain arbitrary text.

- The `WEAK` and `SYNC` keywords are used in error recovery, as discussed in the next section.

- In earlier versions of Coco/R there was a potential pitfall in the specification of attributes. Suppose the urge arises to attribute a *NonTerminal* as follows:

  ```
  SomeNonTerminal< record->field >
  ```

  where the parameter uses the right arrow selection operator "`->`". Since the "`>`" would normally have been taken as a Cocol meta-bracket, this had to be recoded in terms of other operators as

  ```
  SomeNonTerminal< (*record).field >
  ```

  The current versions of Coco/R allow for attributes to be demarcated by "`<.`" and "`.>`" brackets to allow for this situation, and for other operators that involve the `>` character.

- Close perusal of the grammar for *Expression* will reveal that it is legal to write a *Production* in which an *Action* appears to be associated with an alternative for an *Expression* that contains no terminals or non- terminals at all. This feature is often useful. For example we might have

  ```
  Option =   "push" (. stack[++top] = item; .)
           | "pop"  (. item = stack[top--]; .)
           |        (. for (int i = top; i > 0; i--) cout << stack[i]; .) .
  ```

- Another useful feature that can be exploited is the ability of an *Action* to drive the parsing process "semantically". For example, the specification of assignment statements and procedure calls in a simple language might be defined as follows so as to conform to LL(1)

restrictions

```
AssignmentOrCall = Identifier [ ":=" Expression ] .
```

Clearly the semantics of the two statement forms are very different. To handle this we might write the grammar on the lines of

```
AssignmentOrCall
= Identifier<name>          (. Lookup(name);
                               if (IsProcedure(name))
                                 { HandleCall(name); return; } .)
  ":=" Expression<value>    (. HandleAssignment(name, value); .) .
```

12.4.2 Syntax error recovery

Compiler generators vary tremendously in the way in which they provide for recovery from syntactic errors, a subject that was discussed in section 10.3.

The technique described there, although systematically applicable, slows down error-free parsing, inflates the parser code, and is relatively difficult to automate. Coco/R uses a simpler technique, as suggested by Wirth (1986), since this has proved to be almost as effective, and is very easily understood. Recovery takes place only at a rather small number of **synchronization points** in the grammar. Errors at other points are reported, but cause no recovery - parsing simply continues up to the next synchronization point. One consequence of this simplification is that many spurious errors are then likely to be detected for as long as the parser and the input remain out of step. An effective technique for handling this is to arrange that errors are simply not reported if they follow too closely upon one another (that is, a minimum amount of text must be correctly parsed after one error is detected before the next can be reported).

In the simplest approach to using this technique, the designer of the grammar is required to specify synchronization points explicitly. As it happens, this does not usually turn out to be a difficult task: the usual heuristic is to choose locations in the grammar where especially safe terminals are expected that are hardly ever missing or mistyped, or appear so often in source code that they are bound to be encountered again at some stage. In most Pascal-like languages, for example, good candidates for synchronization points are the beginning of a statement (where keywords like `IF` and `WHILE` are expected), the beginning of a declaration sequence (where keywords like `CONST` and `VAR` are expected), or the beginning of a type definition (where keywords like `RECORD` and `ARRAY` are expected).

In Cocol, a synchronization point is specified by the keyword `SYNC`, and the effect is to generate code for a loop that is prepared simply to consume source tokens until one is found that would be acceptable at that point. The sets of such terminals can be precomputed at parser generation time. They are always extended to include the end-of-file symbol (denoted by the keyword `EOF`), thus guaranteeing that if all else fails, synchronization will succeed at the end of the source text.

For our case study we might choose the end of the routine for handling a subtotal as such a point:

```
Subtotal = Range { "+" Range } SYNC ( "accept" | "cancel" ) .
```

This would have the effect of generating code on the following lines:

```
PROCEDURE Subtotal;
  BEGIN
    Range;
    WHILE Sym = plus DO GetSym; Range END;
    WHILE Sym ∉ { accept, cancel, EOF } DO GetSym END;
    IF Sym ∈ { accept, cancel } THEN GetSym END;
```

```
END
```

The union of all the synchronization sets (which we shall denote by *AllSyncs*) is also computed by Coco/R, and is used in further refinements on this idea. A terminal can be designated to be *weak* in a certain context by preceding its appearance in the phrase structure grammar with the keyword `WEAK`. A weak terminal is one that might often be mistyped or omitted, such as the semicolon between statements. When the parser expects (but does not find) such a terminal, it adopts the strategy of consuming source tokens until it recognizes either a legal successor of the weak terminal, or one of the members of *AllSyncs* - since terminals expected at synchronization points are considered to be very "strong", it makes sense that they never be skipped in any error recovery process.

As an example of how this could be used, consider altering our case study grammar to read:

```
Calc     = WEAK "clear" Subtotal { Subtotal } WEAK "total" .
Subtotal = Range { "+" Range } SYNC ( "accept" | "cancel" ) .
Range    = Amount [ ".." Amount ] .
Amount   = number .
```

This would give rise to code on the lines of

```
PROCEDURE Calc;
  BEGIN
    ExpectWeak(clear, FIRST(Subtotal)); (* ie { number } *)
    Subtotal; WHILE Sym = number DO Subtotal END;
    ExpectWeak(total, { EOF })
  END
```

The `ExpectWeak` routine would be internal to the parser, implemented on the lines of:

```
PROCEDURE ExpectWeak (Expected : TERMINAL; WeakFollowers : SYMSET);
  BEGIN
    IF Sym = Expected
      THEN GetSym
      ELSE
        ReportError(Expected);
        WHILE sym ∉ (WeakFollowers + AllSyncs) DO GetSym END
    END
  END
```

Weak terminals give the parser another chance to synchronize in case of an error. The `WeakFollower` sets can be precomputed at parser generation time, and the technique causes no run-time overhead if the input is error-free.

Frequently iterations start with a weak terminal, in situations described by EBNF of the form

```
Sequence =  FirstPart { "WEAK" ExpectedTerminal  IteratedPart } LastPart .
```

Such terminals will be called *weak separators* and can be handled in a special way: if the *ExpectedTerminal* cannot be recognized, source tokens are consumed until a terminal is found that is contained in one of the following three sets:

FOLLOW(*ExpectedTerminal*) (that is, FIRST(*IteratedPart*))
FIRST(*LastPart*)
AllSyncs

As an example of this, suppose we were to modify our case study grammar to read

```
Subtotal = Range { WEAK "+" Range } ( "accept" | "cancel" ) .
```

The generated code would then be on the lines of

```
PROCEDURE Subtotal;
  BEGIN
    Range;
    WHILE WeakSeparator(plus, { number }, { accept, cancel } ) DO
      Range
    END;
    IF Sym ∈ {accept, cancel } THEN GetSym END;
  END
```

The `WeakSeparator` routine would be implemented internally to the parser on the lines of

```
BOOLEAN FUNCTION WeakSeparator (Expected : TERMINAL;
                                WeakFollowers, IterationFollowers : SYMSET);
  BEGIN
    IF Sym = Expected THEN GetSym; RETURN TRUE
      ELSIF Sym ∈ IterationFollowers THEN RETURN FALSE
      ELSE
        ReportError(Expected);
        WHILE Sym ∉ (WeakFollowers + IterationFollowers + AllSyncs) DO
          GetSym
        END;
        RETURN Sym ∈ WeakFollowers
    END
  END
```

Once again, all the necessary sets can be precomputed at generation time. Occasionally, in highly embedded grammars, the inclusion of *AllSyncs* (which tends to be "large") may detract from the efficacy of the technique, but with careful choice of the placing of `WEAK` and `SYNC` keywords it can work remarkably well.

12.4.3 Grammar checks

Coco/R performs several tests to check that the grammar submitted to it is well-formed. In particular it checks that

- each non-terminal has been associated with exactly one production;
- there are no useless productions (in the sense discussed in section 8.3.1);
- the grammar is cycle free (in the sense discussed in section 8.3.3);
- all tokens can be distinguished from one another (that is, no two terminals have been declared to have the same structure).

If any of these tests fail, no code generation takes place. In other respects the system is more lenient. Coco/R issues warnings if analysis of the grammar reveals that

- a non-terminal is nullable (this occurs frequently in correct grammars, but may sometimes be indicative of an error);
- the LL(1) conditions are violated, either because at least two alternatives for a production have FIRST sets with elements in common, or because the FIRST and FOLLOWER sets for a nullable string have elements in common.

If Coco/R reports an LL(1) error for a construct that involves alternatives or iterations, the user should be aware that the generated parser is highly likely to misbehave. As simple examples, productions like the following

```
P = "a" A | "a" B .
Q = [ "c" B ] "c"  .
R = { "d" C } "d" .
```

result in generation of code that can be described algorithmically as

```
IF Sym = "a" THEN Accept("a"); A ELSIF Sym = "a" THEN Accept("a"); B END;
```

```
IF Sym = "c" THEN Accept("c"); B END; Accept("c");
WHILE Sym = "d" DO Accept("d"); C END; Accept("d");
```

Of these, only the second can possibly ever have any meaning (as it does in the case of the "dangling else"). If these situations arise it may often be necessary to redesign the grammar.

12.4.4 Semantic errors

The parsers generated by Coco/R handle the reporting of syntax errors automatically. The default driver programs can summarize these errors at the end of compilation, along with source code line and column references, or produce source code listings with the errors clearly marked with explanatory messages (an example of such a listing appears in section 12.4.7). Pure syntax analysis cannot reveal static semantic errors, but Coco/R supports a mechanism whereby the grammar designer can arrange for such errors to be reported in the same style as is used for syntactic errors. The parser class has routines that can be called from within the semantic actions, with an error number parameter that can be associated with a matching user-defined message.

In the grammar of our case study, for example, it might make sense to introduce a semantic check into the actions for the non-terminal `Range`. The grammar allows for a range of values to be summed; clearly this will be awkward if the "upper" limit is supplied as a lower value than the "lower" limit. The code below shows how this could be detected, resulting in the reporting of the semantic error 200.

```
Range<double &r>
=                          (. double low, high; .)
  Amount<low>              (. r = low; .)
  [ ".." Amount<high>      (. if (low > high) SemError(200);
                              else while (low < high) { low++; r += low; } .)
  ] .
```

(Alternatively, we could also arrange for the system to run the loop in the appropriate direction, and not regard this as an error at all.) Numbers chosen for semantic error reporting must start at some fairly large number to avoid conflict with the low numbers chosen internally by Coco/R to report syntax errors.

12.4.5 Interfacing to support modules

It will not have escaped the reader’s attention that the code specified in the actions of the attributed grammar will frequently need to make use of routines that are not defined by the grammar itself. Two typical situations are exemplified in our case study.

Firstly, it has seen fit to make use of the `printf` routine from the `stdio` library found in all standard C and C++ implementations. To make use of such routines - or ones defined in other support libraries that the application may need - it is necessary simply to incorporate the appropriate `#define, IMPORT` or `USES` clauses into the grammar before the scanner specification, as discussed in section 12.2.

Secondly, the need arises in routines like `Amount` to be able to convert a string, recognized by the scanner as a number, into a numerical value that can be passed back via a formal parameter to the calling routine (`Range`). This situation arises so frequently that the parser interface defines several routines to simplify the extraction of this string. The production for `Amount`, when fully attributed, might take the form

```
Amount<double &a>
= number                   (. char str[100];
                              LexString(str, 100);
```

```
                                a = atof(str); .) .
```

The `LexString` routine (defined in the parser interface) retrieves the string into the local string `str`, whence it is converted to the `double` value `a` by a call to the `atof` function that is defined in the `stdlib` library. If the functionality of routines like `LexString` and `LexName` is inadequate, the user can incorporate calls to the even lower level routines defined in the scanner interface, such as were mentioned in section 12.3.6.

12.4.6 The parser interface

The parser generated by Coco/R defines various routines that may be called from an application. As for the scanner, the form of the interface depends on the host system. For the C++ version, it effectively takes the form below. (As before, there is actually an underlying class hierarchy, and the declarations are really slightly different from those presented here).

The functionality provides for the parser to

- initiate the parse for the goal symbol by calling `Parse()`.
- investigate whether the parse succeeded by calling `Successful()`.
- report on the presence of syntactic and semantic errors by calling `SynError` and `SemError`.
- obtain the lexeme value of a particular token in one of four ways (`LexString`, `LexName`, `LookAheadString` and `LookAheadName`). Calls to `LexString` are most common; the others are used for special variations.

```
class grammarParser
{ public:
    grammarParser(AbsScanner *S, CRError *E);
    // Constructs parser associated with scanner S and error reporter E

    void Parse();
    // Parses the source

    int Successful();
    // Returns 1 if no errors have been recorded while parsing

  private:
    void LexString(char *lex, int size);
    // Retrieves at most size characters from the most recently parsed
    // token into lex

    void LexName(char *lex, int size);
    // Retrieves at most size characters from the most recently parsed
    // token into lex, converted to upper case if IGNORE CASE was specified

    void LookAheadString(char *lex, int size);
    // Retrieves at most size characters from the lookahead token into lex

    void LookAheadName(char *lex, int size);
    // Retrieves at most size characters from the lookahead token into lex,
    // converted to upper case if IGNORE CASE was specified

    void SynError(int errorcode);
    // Reports syntax error denoted by errorcode

    void SemError(int errorcode);
    // Reports semantic error denoted by errorcode

    // ... Prototypes of functions for parsing each non-terminal in grammar
};
```

12.4.7 A complete example

To place all of the ideas of the last sections in context, we present a complete version of the attributed grammar for our case study:

```
$CX     /* pragmas - generate compiler, and use C++ classes */
```

```
COMPILER Calc

  #include <stdio.h>
  #include <stdlib.h>

  CHARACTERS
    digit =  "0123456789" .

  IGNORE CHR(9) .. CHR(13)

  TOKENS
    number =   digit { digit } [ "." digit { digit } ]
             | digit { digit } CONTEXT ( ".." ) .

  PRAGMAS
    page   = "page" .        (. printf("\f"); .)

  PRODUCTIONS
    Calc
    =                        (. double total = 0.0, sub; .)
      WEAK "clear"
      { Subtotal<sub>        (. total += sub; .)
      } SYNC "total"         (. printf("   total: %5.2f\n", total); .)
      .

    Subtotal<double &s>
    =                        (. double r; .)
      Range<s>
      { WEAK "+" Range<r>    (. s += r; .)
      } SYNC
      (   "accept"           (. printf("subtotal: %5.2f\n", s); .)
        | "cancel"           (. s = 0.0; .)
      ) .

    Range<double &r>
    =                        (. double low, high; .)
      Amount<low>            (. r = low; .)
      [ ".." Amount<high>    (. if (low > high) SemError(200);
                                else while (low < high)
                                { low++; r += low; } .)
      ] .

    Amount<double &a>
    = number                 (. char str[100];
                                LexString(str, 100);
                                a = atof(str); .) .

END Calc.
```

To show how errors are reported, we show the output from applying the generated system to input that is fairly obviously incorrect.

```
    1  clr
***** ^  clear  expected (E2)
    2  1 + 2 + 3 .. 4 + 4..5 accep
*****                       ^  +  expected (E4)
    3  3.4 5 cancel
*****      ^  +  expected (E4)
    4  3 + 4 .. 2 + 6 accept
*****           ^ High < Low (E200)
    5  TOTAL
*****  ^ unexpected symbol in Calc (E10)
```

12.5 The driver program

The most important tasks that Coco/R has to perform are the construction of the scanner and parser. However, these always have to be incorporated into a complete program before they become useful.

12.5.1 Essentials of the driver program

Any main routine for a driver program must be a refinement of ideas that can be summarized:

```
BEGIN
  Open(SourceFile);
  IF Okay THEN
    InstantiateScanner;
    InstantiateErrorHandler;
    InstantiateParser;
    Parse();
    IF Successful() THEN ApplicationSpecificAction END
  END
END
```

Much of this can be automated, of course, and Coco/R can generate such a program, consistent with its other components. To do so requires the use of an appropriate frame file. A generic version of this is supplied with the distribution. Although it may be suitable for constructing simple prototypes, it acts best as a model from which an application-specific frame file can easily be derived.

12.5.2 Customizing the driver frame file

A customized driver frame file generally requires at least three simple additions:

- It is often necessary to declare global or external variables, and to add application specific `#include, USES` or `IMPORT` directives so that the necessary library support will be provided.

- The section dealing with error messages may need extension if the grammar has made use of the facility for adding errors to those derived by the parser generator, as discussed in section 12.4.4. For example, the default C++ driver frame file has code that reads

  ```
  char *MyError::GetUserErrorMsg(int n)
  { switch (n) {
      // Put your customized messages here
      default:  return "Unknown error";
    }
  }
  ```

  To tailor this to the case study application we should need to add an option to the `switch` statement:

  ```
  char *MyError::GetUserErrorMsg(int n)
  { switch (n) {
      case 200: return "High < Low";
      default:  return "Unknown error";
    }
  }
  ```

- Finally, at the end of the default frame file can be found code like

  ```
   // instantiate Scanner, Parser and Error handler
   Scanner = new -->ScanClass(S_src, -->IgnoreCase);
   Error   = new MyError(SourceName, Scanner);
   Parser  = new -->ParserClass(Scanner, Error);

   // parse the source
   Parser->Parse();
   close(S_src);

   // Add to the following code to suit the application
   if (Error->Errors) fprintf(stderr, "Compilation errors\n");
   if (Listinfo) SourceListing(Error, Scanner);
   else if (Error->Errors) Error->SummarizeErrors();

   delete Scanner;
   delete Parser;
   delete Error;
  }
  ```

  the intention of which should be almost self explanatory. For example, in the case of a

compiler/interpreter such as we shall discuss in a later chapter, we might want to modify this to read

```
// generate source listing
FILE *lst = fopen("listing");
Error->SetOutput(lst);
Error->PrintListing(Scanner);
fclose(lst);

if (Error->Errors)
  fprintf(stderr, "Compilation failed - see %s\n", ListName);
else {
  fprintf(stderr, "Compilation successful\n");
  CGen->getsize(codelength, initsp);
  Machine->interpret(codelength, initsp);
}
```

Exercises

12.1 Study the code produced by Coco/R from the grammar used in this case study. How closely does it correspond to what you might have written by hand?

12.2 Experiment with the grammar suggested in the case study. What happens if the `CONTEXT` clause is omitted in the scanner specification? What happens if the placement of the `WEAK` and `SYNC` keywords is changed?

12.3 Extend the system in various ways. For example, direct output to a file other than `stdout`, use the `iostreams` library rather than the `stdio` library, develop the actions so that they conform to "traditional" C (rather than using reference parameters), or arrange that ranges can be correctly interpreted in either order.

Further reading

The text by Rechenberg and Mössenböck (1989) describes the original Coco system in great detail. This system did not have an integrated scanner generator, but made use of one known as Alex (Mössenböck, 1986). Dobler and Pirklbauer (1990) and Dobler (1991) discuss Coco-2, a variant of Coco that incorporated automatic and sophisticated error recovery into table-driven LL(1) parsers. Literature on the inner workings of Coco/R is harder to come by, but the reader is referred to the papers by Mössenböck (1990a, 1990b).

13 USING COCO/R - CASE STUDIES

The best way to come to terms with the use of a tool like Coco/R is to try to use it, so in this chapter we make use of several case studies to illustrate how simple and powerful a tool it really is.

13.1 Case study - Understanding C declarations

It is generally acknowledged, even by experts, that the syntax of declarations in C and C++ can be quite difficult to understand. This is especially true for programmers who have learned Pascal or Modula-2 before turning to a study of C or C++. Simple declarations like

```
int x, list[100];
```

present few difficulties (`x` is a scalar integer, `list` is an array of 100 integers). However, in developing more abstruse examples like

```
char **a;      // a is a pointer to a pointer to a character
int *b[10];    // b is an array of 10 pointers to single integers
int (*c)[10];  // c is a pointer to an array of 10 integers
double *d();   // d is a function returning a pointer to a double
char (*e)();   // e is a pointer to a function returning a character
```

it is easy to confuse the placement of the various brackets, parentheses and asterisks, perhaps even writing syntactically correct declarations that do not mean what the author intended. By the time one is into writing (or reading) declarations like

```
short (*(*f())[])();
double (*(*g[50])())[15];
```

there may be little consolation to be gained from learning that C was designed so that the syntax of declarations (defining occurrences) should mirror the syntax for access to the corresponding quantities in expressions (applied occurrences).

Algorithms to help humans unravel such declarations can be found in many text books - for example, the recent excellent one by King (1996), or the original description of C by Kernighan and Ritchie (1988). In this latter book can be found a hand-crafted recursive descent parser for converting a subset of the possible declaration forms into an English description. Such a program is very easily specified in Cocol.

The syntax of the restricted form of declarations that we wish to consider can be described by

```
Decl      =  { name Dcl ";" } .
Dcl       =  { "*" } DirectDcl .
DirectDcl =    name
             | "(" Dcl ")"
             | DirectDcl "(" ")"
             | DirectDcl "[" [ number ] "]" .
```

if we base the productions on those found in the usual descriptions of C, but change the notation to match the one we have been using in this book. Although these productions are not in LL(1) form, it is easy to find a way of eliminating the troublesome left recursion. It also turns out to be expedient to rewrite the production for *Dcl* so as to use right recursion rather than iteration:

```
Decl      =  { name Dcl ";" } .
```

```
Dcl       =  "*" Dcl | DirectDcl .
DirectDcl =  ( name | "(" Dcl ")" ) { Suffix } .
Suffix    =  "(" ")" | "[" [ number ] "]" .
```

When adding attributes we make use of ideas similar to those already seen for the conversion of infix expressions into postfix form in section 11.1. We arrange to read the token stream from left to right, writing descriptions of some tokens immediately, but delaying the output of descriptions of others. The full Cocol specification follows readily as

```
$CX   /* Generate Main Module, C++ */
COMPILER Decl
#include <stdlib.h>
#include <iostream.h>

CHARACTERS
  digit =  "0123456789" .
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyx_" .

IGNORE CHR(9) .. CHR(13)

TOKENS
  number = digit { digit } .
  name = letter { letter } .

PRODUCTIONS
  Decl
  =                     (. char Tipe[100]; .)
    { name              (. LexString(Tipe, sizeof(Tipe) - 1); .)
      Dcl               (. cout << ' ' << Tipe << endl; .)
      ";" } .

  Dcl
  =   "*" Dcl           (. cout << " pointer to"; .)
    | DirectDcl .

  DirectDcl
  =                     (. char Name[100]; .)
    (   name            (. LexString(Name, sizeof(Name) - 1);
                           cout << ' ' << Name << " is"; .)
      | "(" Dcl ")"
    ) { Suffix } .

  Suffix
  =                     (. char buff[100]; .)
      "["               (. cout << " array ["; .)
       [ number         (. LexString(buff, sizeof(buff) - 1);
                           cout << atoi(buff); .)
       ]
      "]"               (. cout << "] of"; .)
    | "(" ")"           (. cout << " function returning"; .) .

END Decl.
```

Exercises

13.1 Perusal of the original grammar (and of the equivalent LL(1) version) will suggest that the following declarations would be allowed. Some of them are, in fact, illegal in C:

```
int f()[100];  // Functions cannot return arrays
int g()();     // Functions cannot return functions
int x[100]();  // We cannot declare arrays of functions
int p[12][20]; // We are allowed arrays of arrays
int q[][100];  // We are also allowed to declare arrays like this
int r[100][];  // We are not allowed to declare arrays like this
```

Can you write a Cocol specification for a parser that accepts only the valid combinations of suffixes? If not, why not?

13.2 Extend the grammar to cater for the declaration of more than one item based on the same type, as exemplified by

```
int f[100], *x, (*g)[100];
```

13.3 Extend the grammar and the parser to allow function prototypes to describe parameter lists, and to allow variable declarators to have initializers, as exemplified by

```
int x = 10, y[3] = { 4, 5, 6 };
int z[2][2] = {{ 4, 5 }, { 6, 7 }};
double f(int x, char &y, double *z);
```

13.4 Develop a system that will do the reverse operation - read in a description of a declaration (such as might be output from the program we have just discussed) and construct the C code that corresponds to this.

13.2 Case study - Generating one-address code from expressions

The simple expression grammar is, understandably, very often used in texts on programming language translation. We have already seen it used as the basis of a system to convert infix to postfix (section 11.1), and for evaluating expressions (section 11.2). In this case study we show how easy it is to attribute the grammar to generate one- address code for a multi-register machine whose instruction set supports the following operations:

```
LDI Rx,value     ;  Rx := value (immediate)
LDA Rx,variable  ;  Rx := value of variable (direct)
ADD Rx,Ry        ;  Rx := Rx + Ry
SUB Rx,Ry        ;  Rx := Rx - Ry
MUL Rx,Ry        ;  Rx := Rx * Ry
DVD Rx,Ry        ;  Rx := Rx / Ry
```

For this machine we might translate some example expressions into code as follows:

```
a + b       5 * 6       x / 12       (a + b) * (c - 5)

LDA R1,a    LDI R1,5    LDA R1,x     LDA  R1,a    ;  R1 := a
LDA R2,b    LDI R2,6    LDI R2,12    LDA  R2,b    ;  R2 := b
ADD R1,R2   MUL R1,R2   DVD R1,R2    ADD  R1,R2   ;  R1 := a+b
                                     LDA  R2,c    ;  R2 := c
                                     LDI  R3,5    ;  R3 := 5
                                     SUB  R2,R3   ;  R2 := c-5
                                     MUL  R1,R2   ;  R1 := (a+b)*(c-5)
```

If we make the highly idealized assumption that the machine has an inexhaustible supply of registers (so that any values may be used for *x* and *y*), then an expression compiler becomes almost trivial to specify in Cocol.

```
$CX /* Compiler, C++ */
COMPILER Expr

CHARACTERS
  digit    = "0123456789" .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

IGNORE CHR(9) .. CHR(13)

TOKENS
  number   = digit { digit } .
  variable = letter .

PRODUCTIONS
  Expr
  = { Expression<1> SYNC ";"   (. printf("\n"); .)
    } .

  Expression<int R>
  = Term<R>
    {   "+" Term<R+1>          (. printf("ADD R%d,R%d\n", R, R+1); .)
```

```
      | "-" Term<R+1>          (. printf("SUB R%d,R%d\n", R, R+1); .)
    } .

  Term<int R>
  = Factor<R>
    {   "*" Factor<R+1>        (. printf("MUL R%d,R%d\n", R, R+1); .)
      | "/" Factor<R+1>        (. printf("DVD R%d,R%d\n", R, R+1); .)
    } .

  Factor<int R>
  =                            (. char CH; int N; .)
      Identifier<CH>           (. printf("LDA R%d,%c\n", R, CH); .)
    | Number<N>                (. printf("LDI R%d,%d\n", R, N); .)
    | "(" Expression<R> ")" .

  Identifier<char &CH>
  =  variable                  (. char str[100];
                                  LexString(str, sizeof(str) - 1);
                                  CH = str[0]; .) .

  Number<int &N>
  =  number                    (. char str[100];
                                  LexString(str, sizeof(str) - 1);
                                  N = atoi(str); .) .

END Expr.
```

The formal attribute to each routine is the number of the register in which the code generated by that routine is required to store the value for whose computation it is responsible. Parsing starts by assuming that the final value is to be stored in register 1. A binary operation is applied to values in registers *x* and *x* + 1, leaving the result in register *x*. The grammar is factorized, as we have seen, in a way that correctly reflects the associativity and precedence of the parentheses and arithmetic operators as they are found in infix expressions, so that, where necessary, the register numbers increase steadily as the parser proceeds to decode complex expressions.

Exercises

13.5 Use Coco/R to develop a program that will convert infix expressions to postfix form.

13.6 Use Coco/R to develop a program that will evaluate infix arithmetic expressions directly.

13.7 The parser above allows only single character variable names. Extend it to allow variable names that consist of an initial letter, followed by any sequence of digits and letters.

13.8 Suppose that we wished to be able to generate code for expressions that permit leading signs, as for example + *x* * (- *y* + *z*). Extend the grammar to describe such expressions, and then develop a program that will generate appropriate code. Do this in two ways (a) assume that there is no special machine instruction for negating a register (b) assume that such an operation is available (`NEG Rx`).

13.9 Suppose the machine also provided logical operations:

```
AND Rx,Ry        ;  Rx := Rx AND Ry
OR  Rx,Ry        ;  Rx := Rx OR Ry
XOR Rx,Ry        ;  Rx := Rx XOR Ry
NOT Rx           ;  Rx := NOT Rx
```

Extend the grammar to allow expressions to incorporate infix and prefix logical operations, in addition to arithmetic operations, and develop a program to translate them into simple machine code. This will require some decision as to the relative precedence of all the operations. NOT always takes precedence over AND, which in turn takes precedence over OR. In Pascal and

Modula-2, NOT, AND and OR are deemed to have precedence equal to unary negation, multiplication and addition (respectively). However, in C and C++, NOT has precedence equal to unary negation, while AND and OR have lower precedence than the arithmetic operators - the 16 levels of precedence in C, like the syntax of declarations, are another example of baroque language design that cause a great difficulty to beginners. Choose whatever relative precedence scheme you prefer, or better still, attempt the exercise both ways.

13.10 (Harder). Try to incorporate short-circuit Boolean semantics into the language suggested by Exercise 13.9, and then use Coco/R to write a translator for it. The reader will recall that these semantics demand that

```
A AND B   is defined to mean   IF A THEN B ELSE FALSE
A OR  B   is defined to mean   IF A THEN TRUE ELSE B
```

that is to say, in evaluating the AND operation there is no need to evaluate the second operand if the first one is found to be FALSE, and in evaluating the OR operation there is no need to evaluate the second operand if the first is found to be TRUE. You may need to extend the instruction set of the machine to provide conditional and other branch instructions; feel free to do so!

13.11 It is unrealistic to assume that one can simply allocate registers numbered from 1 upwards. More usually a compiler has to select registers from a set of those known to be free at the time the expression evaluation commences, and to arrange to release the registers once they are no longer needed for storing intermediate values. Modify the grammar (and hence the program) to incorporate this strategy. Choose a suitable data structure to keep track of the set of available registers - in Pascal and Modula-2 this becomes rather easy; in C++ you could make use of the template class for set handling discussed briefly in section 10.3.

13.12 It is also unreasonable to assume that the set of available registers is inexhaustible. What sort of expression requires a large set of registers before it can be evaluated? How big a set do you suppose is reasonable? What sort of strategy do you suppose has to be adopted if a compiler finds that the set of available registers becomes exhausted?

13.3 Case study - Generating one-address code from an AST

It should not take much imagination to realize that code generation for expression evaluation using an "on-the fly" technique like that suggested in section 13.2, while easy, leads to very inefficient and bloated code - especially if, as is usually the case, the machine instruction set incorporates a wider range of operations. If, for example, it were to include direct and immediate addressing operations like

```
ADD Rx,variable  ;  Rx := Rx + value of variable
SUB Rx,variable  ;  Rx := Rx - value of variable
MUL Rx,variable  ;  Rx := Rx * value of variable
DVD Rx,variable  ;  Rx := Rx / value of variable

ADI Rx,constant  ;  Rx := Rx + value of constant
SBI Rx,constant  ;  Rx := Rx - value of constant
MLI Rx,constant  ;  Rx := Rx * value of constant
DVI Rx,constant  ;  Rx := Rx / value of constant
```

then we should be able to translate the examples of code shown earlier far more effectively as follows:

```
a + b       5 * 6       x / 12       (a + b) * (c - 5)

LDA R1,a    LDI R1,30   LDA R1,x     LDA  R1,a    ;  R1 := a
ADD R1,b                DVI R1,12    ADD  R1,b    ;  R1 := a + b
                                     LDA  R2,c    ;  R2 := c
                                     SBI  R2,5    ;  R2 := c - 5
                                     MUL  R1,R2   ;  R1 := (a+b)*(c-5)
```

To be able to generate such code requires that we delay the choice of instruction somewhat - we should no longer simply emit instructions as soon as each operator is recognized (once again we can see a resemblance to the conversion from infix to postfix notation). The usual strategy for achieving such optimizations is to arrange to build an abstract syntax tree (AST) from the expression, and then to "walk" it in LRN (post) order, emitting machine code apposite to the form of the operation associated with each node. An example may make this clearer. The tree corresponding to the expression (*a* + *b*) * (*c* - 5) is shown in Figure 13.1.

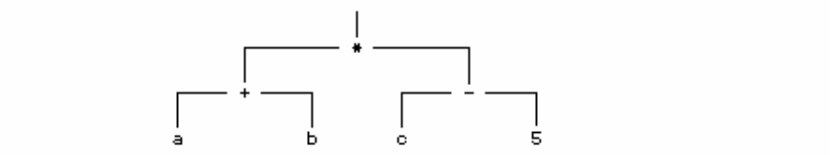


Figure 13.1 An AST corresponding to the expression (a + b) * (c - 5)

The code generating operations needed as each node is visited are depicted in Figure 13.2.

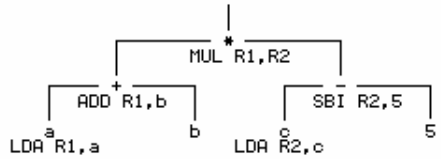


Figure 13.2 Code generation needed when visiting each node in an AST

It is, in fact, remarkably easy to attribute our grammar so as to incorporate tree-building actions instead of immediate code generation:

```
$CX /* Compiler, C++ */
COMPILER Expr
/* Convert infix expressions into machine code using a simple AST */

#include "trees.h"

CHARACTERS
  digit  =  "0123456789" .
  letter =  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

IGNORE CHR(9) .. CHR(13)

TOKENS
  number = digit { digit } .
  variable = letter .

PRODUCTIONS
  Expr
  =                           (. AST Exp; .)
    { Expression<Exp>
      SYNC ";"                (. if (Successful()) GenerateCode(Exp); .)
    } .

  Expression<AST &E>
  =                           (. AST T; .)
    Term<E>
    {   "+" Term<T>           (. E = BinOpNode(Plus, E, T); .)
      | "-" Term<T>           (. E = BinOpNode(Minus, E, T); .)
    } .
```

```
  Term<AST &T>
  =                          (. AST F; .)
    Factor<T>
    {   "*" Factor<F>        (. T = BinOpNode(Times, T, F); .)
      | "/" Factor<F>        (. T = BinOpNode(Slash, T, F); .)
    } .

  Factor<AST &F>
  =                          (. char CH; int N; .)
                             (. F = EmptyNode(); .)
  (   Identifier<CH>         (. F = VarNode(CH); .)
    | Number<N>              (. F = ConstNode(N); .)
    | "(" Expression<F> ")"
  ) .

  Identifier<char &CH>
  =  variable                (. char str[100];
                                LexName(str, sizeof(str) - 1);
                                CH = str[0]; .) .

  Number<int &N>
  =  number                  (. char str[100];
                                LexString(str, sizeof(str) - 1);
                                N = atoi(str); .) .

END Expr.
```

Here, rather than pass register indices as "value" parameters to the various parsing routines, we arrange that they each return an AST (as a "reference" parameter) - essentially a pointer to a structure created as each *Expression*, *Term* or *Factor* is recognized. The *Factor* parser is responsible for creating the leaf nodes, and these are stitched together to form larger trees as a result of the iteration components in the *Expression* and *Term* parsers. Once the tree has been built in this way - that is, after the goal symbol has been completely parsed - we can walk it so as to generate the code.

The reader may feel a bit cheated, as this does not reveal very much about how the trees are really constructed. However, that is in the spirit of "data abstraction"! The grammar above can be used unaltered with a variety of implementations of the AST tree handling module. In compiler technology terminology, we have succeeded in separating the "front end" or parser from the "back end" or tree-walker that generates the code. By providing machine specific versions of the tree-walker we can generate code for a variety of different machines, indulge in various optimization techniques, and so on. The AST tree-builder and tree-walker have the following interface:

```
enum optypes { Load, Plus, Minus, Times, Slash };

class NODE;

typedef NODE* AST;

AST BinOpNode(optypes op, AST left, AST right);
// Creates an AST for the binary operation "left op right"

AST VarNode(char name);
// Creates an AST for a variable factor with specified name

AST ConstNode(int value);
// Creates an AST for a constant factor with specified value

AST EmptyNode();
// Creates an empty node

void GenerateCode (AST A);
// Generates code from AST A
```

Here we are defining an `AST` type as a pointer to a (dynamically allocated) `NODE` object. The functions exported from this interface allow for the construction of several distinct varieties of nodes, of course, and in particular (a) an "empty" node (b) a "constant" node (c) a "variable" node and (d) a "binary operator" node. There is also a routine that can walk the tree, generating code as

each node is visited.

In traditional implementations of this module we should have to resort to constructing the `NODE` type as some sort of variant record (in Modula-2 or Pascal terminology) or union (in C terminology), and on the source diskette can be found examples of such implementations. In languages that support object-oriented programming it makes good sense to define the `NODE` type as an abstract base class, and then to derive the other types of nodes as sub- classes or derived classes of this type. The code below shows one such implementation in C++ for the generation of code for our hypothetical machine. On the source diskette can be found various class based implementations, including one that generates code no more sophisticated than was discussed in section 13.2, as well as one matching the same interface, but which generates code for the single-accumulator machine introduced in Chapter 4. There are also equivalent implementations that make use of the object-oriented extensions found in Turbo Pascal and various dialects of Modula-2.

```
// Abstract Syntax Tree facilities for simple expression trees
// used to generate reasonable one-address machine code.

#include <stdio.h>
#include "trees.h"

class NODE
{ friend AST BinOpNode(optypes op, AST left, AST right);
  friend class BINOPNODE;
  public:
    NODE()                              { defined = 0; }
    virtual void load(int R) = 0;
    // Generate code for loading value of a node into register R
  protected:
    int value;     // value derived from this node
    int defined;   // 1 if value is defined
    virtual void operation(optypes O, int R) = 0;
    virtual void loadreg(int R)        {;}
};

class BINOPNODE : public NODE
{ public:
    BINOPNODE(optypes O, AST L, AST R)   { op = O; left = L; right = R; }
    virtual void load(int R);
  protected:
    optypes op;
    AST left, right;
    virtual void operation(optypes O, int R);
    virtual void loadreg(int R)        { load(R); }
};

void BINOPNODE::operation(optypes op, int R)
{ switch (op)
  { case Load:  printf("LDA"); break;
    case Plus:  printf("ADD"); break;
    case Minus: printf("SUB"); break;
    case Times: printf("MUL"); break;
    case Slash: printf("DVD"); break;
  }
  printf(" R%d,R%d\n", R, R + 1);
}

void BINOPNODE::load(int R)
{ if (!left || !right) return;
  left->load(R); right->loadreg(R+1); right->operation(op, R);
  delete left; delete right;
}

AST BinOpNode(optypes op, AST left, AST right)
{ if (left && right && left->defined && right->defined)
  { // constant folding
    switch (op)
    { case Plus:  left->value += right->value; break;
      case Minus: left->value -= right->value; break;
      case Times: left->value *= right->value; break;
      case Slash: left->value /= right->value; break;
    }
    delete right; return left;
  }
  return new BINOPNODE(op, left, right);
```

```
}

class VARNODE : public NODE
{  public:
    VARNODE(char C)                   { name = C; }
    virtual void load(int R)          { operation(Load, R); }
  protected:
    char name;
    virtual void operation(optypes O, int R);
};

void VARNODE::operation(optypes op, int R)
{ switch (op)
  { case Load:  printf("LDA"); break;
    case Plus:  printf("ADD"); break;
    case Minus: printf("SUB"); break;
    case Times: printf("MUL"); break;
    case Slash: printf("DVD"); break;
  }
  printf(" R%d,%c\n", R, name);
}

AST VarNode(char name)
{ return new VARNODE(name); }

class CONSTNODE : public NODE
{ public:
    CONSTNODE(int V)                  { value = V; defined = 1; }
    virtual void load(int R)          { operation(Load, R); }
  protected:
    virtual void operation(optypes O, int R);
};

void CONSTNODE::operation(optypes op, int R)
{ switch (op)
  { case Load:  printf("LDI"); break;
    case Plus:  printf("ADI"); break;
    case Minus: printf("SBI"); break;
    case Times: printf("MLI"); break;
    case Slash: printf("DVI"); break;
  }
  printf(" R%d,%d\n", R, value);
}

AST ConstNode(int value)
{ return new CONSTNODE(value); }

AST EmptyNode()
{ return NULL; }

void GenerateCode(AST A)
{ A->load(1); printf("\n"); }
```

The reader's attention is drawn to several points that might otherwise be missed:

- We have deliberately chosen to implement a single `BINOPNODE` class, rather than using this as a base class from which were derived `ADDNODE, SUBNODE, MULNODE` and `DIVNODE` classes. The alternative approach makes for a useful exercise for the reader.

- When the `BinOpNode` routine constructs a binary node, some optimization is attempted. If both the left and right subexpressions are `defined`, that is to say, are represented by constant nodes, then arithmetic can be done immediately. This is known as **constant folding**, and, once again, is something that is far more easily achieved if an AST is constructed, rather than resorting to "on-the-fly" code generation. It often results in a saving of registers, and in shorter (and hence faster) object code.

- Some care must be taken to ensure that the integrity of the AST is preserved even if the source expression is syntactically incorrect. The *Factor* parser is arranged so as to return an empty node if it fails to recognize a valid member of FIRST(*Factor*), and there are various other checks in the code to ensure that tree walking is not attempted if such nodes have been incorporated into the tree (for example, in the `BINOPNODE::load` and `BinOpNode` routines).

Exercises

13.13 The constant folding demonstrated here is dangerous, in that it has assumed that arithmetic overflow will never occur. Try to improve it.

13.14 One disadvantage of the approach shown here is that the operators have been "hard wired" into the `optypes` enumeration. Extending the parser to handle other operations (such as AND and OR) would require modification in several places, which would be error-prone, and not in the spirit of extensibility that OOP techniques are meant to provide. If this strikes you as problematic, rework the `AST` handler to introduce further classes derived from `BINOPNODE`.

13.15 The tree handler is readily extended to perform other simple optimizations. For example, binary expressions like $x * 1$, $1 * x$, $x + 0$, $x * 0$ are quite easily detected, and the otherwise redundant operations can be eliminated. Try to incorporate some of these optimizations into the routines given earlier. Is it better to apply them while the tree is under construction, or when it is later walked?

13.16 Rework Exercises 13.8 through 13.12 to use abstract syntax trees for intermediate representations of source expressions.

13.4 Case study - How do parser generators work?

Our last case study aims to give the reader some insight into how a program like Coco/R might itself be developed. In effect, we wish to be able to develop a program that will take as input an LL(1) type grammar, and go on to construct a parser for that grammar. As we have seen, such grammars can be described in EBNF notation, and the same EBNF notation can be used to describe itself, rather simply, and in a form suitable for top- down parsing. In particular we might write

```
Syntax       =   { Production } "EOG" .
Production   =   NonTerminal "=" Expression "." .
Expression   =   Term { "|" Term } .
Term         =   [ Factor { Factor } ] .
Factor       =      NonTerminal
                |   Terminal
                |   "(" Expression ")" | "[" Expression "]"
                |   "{" Expression "}" .
```

where *NonTerminal* and *Terminal* would be chosen from a particular set of symbols for a grammar, and where the terminal "`EOG`" has been added to ease the task of recognizing the end of the grammar. It is left to the reader formally to show that this grammar is LL(1), and hence capable of being parsed by recursive descent.

A parser generator may be constructed by enriching this grammar, providing actions at appropriate points so as to construct, from the input data, some code (or similar structure which can later be "executed") either to parse other programs, or to construct parsers for those programs. One method of doing this, outlined by Wirth (1976b, 1986) and Rechenberg and Mössenböck (1989), is to develop the parser actions so that they construct a data structure that encapsulates a syntax diagram representation of the grammar as a graph, and then to apply a graph walker that traverses these syntax diagrams.

To take a particular example, consider the *ClassList* grammar of section 11.5, for which the productions are

```
ClassList  =  ClassName [ Group { ";" Group } ]  "." .
Group      =  Degree ":"  Student { "," Student } .
Degree     =  "BSc" | "BScS" .
ClassName  =  identifier .
Student    =  identifier .
```

A corresponding set of syntax diagrams for these productions is shown in Figure 13.3.

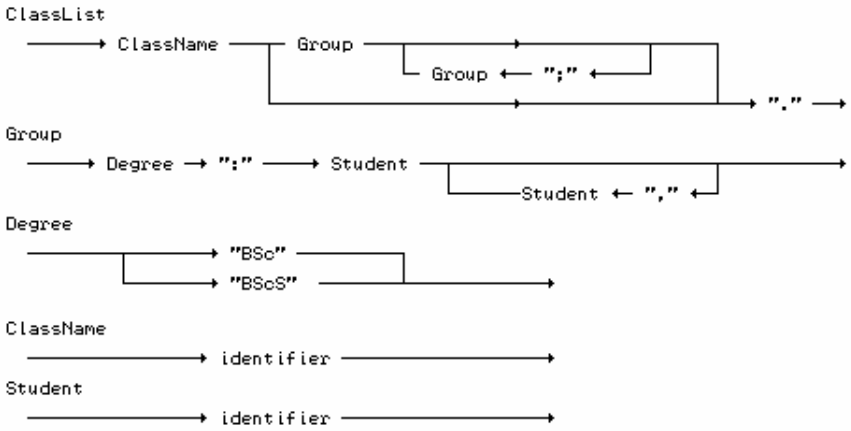


Figure 13.3 Syntax diagrams describing the Classlist grammar

Such graphs may be represented in programs by linked data structures. At the top level we maintain a linked list of nodes, each one corresponding to a non-terminal symbol of the grammar. For each such symbol in the grammar we then go on to introduce (for each of its alternative productions) a sub-graph of nodes linked together.

In these dependent graphs there are two basic types of nodes: those corresponding to terminal symbols, and those corresponding to non-terminals. Terminal nodes can be labelled by the terminal itself; non-terminal nodes can contain pointers back to the nodes in the non-terminal list. Both variants of graph nodes contain two pointers, one (`Next`) designating the symbol that follows the symbol "stored" at the node, and the other (`Alternate`) designating the next in a list of alternatives. Once again, the reader should be able to see that this lends itself to the fruitful adoption of OOP techniques - an abstract base class can be used for a node, with derived classes to handle the specializations.

As it turns out, one needs to take special cognizance of the empty terminal ε, especially in those situations where it appears implicitly through the "{" *Expression* "}" or "[" *Expression* "]" construction rather than through an explicit empty production.

The way in which the graphs are constructed is governed by four quite simple rules:

- A sequence of *Factors* generated by a *Term* gives rise to a list of nodes linked by their `Next` pointers, as shown in Figure 13.4(a);
- A succession of alternative *Terms* produced by an *Expression* gives rise to a list of nodes linked by their `Alternate` pointers, as shown in Figure 13.4(b);
- A loop produced by a factor of the form { *Expression* } gives rise to a structure of the form

shown in Figure 13.4(c);

- An option produced by a factor of the form [*Expression*] gives rise to a structure of the form shown in Figure 13.4(d).

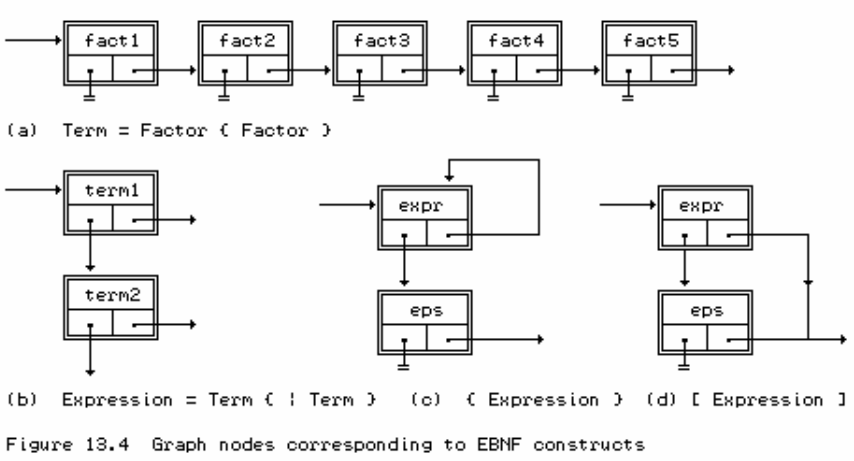


Figure 13.4 Graph nodes corresponding to EBNF constructs

As a complete example, the structures that correspond to our *ClassList* example lead to the graph depicted in Figure 13.5.

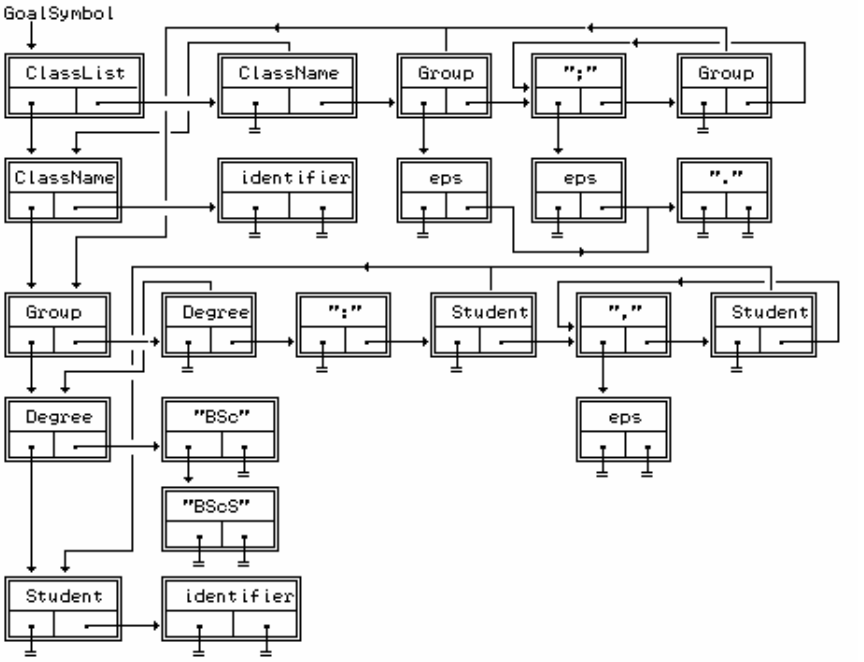


Figure 13.5 Graph depicting the ClassList grammar

Construction of the data structures is a non-trivial exercise - especially when they are extended further to allow for semantic attributes to be associated with the various nodes. As before, we have attempted to introduce a large measure of abstraction in the attributed Cocol grammar given below:

```
$CX /* compiler, C++ */
COMPILER EBNF
/* Augmented Coco/R grammar describing a set of EBNF productions
   and allowing the construction of a graph driven parser */

#include "misc.h"
#include "gp.h"
```

```
extern GP *GParser;

CHARACTERS
  cr       = CHR(13) .
  lf       = CHR(10) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  digit    = "0123456789" .
  noquote1 = ANY - "'" - cr - lf .
  noquote2 = ANY - '"' - cr - lf .

IGNORE CHR(9) .. CHR(13)
IGNORE CASE

COMMENTS FROM "(*" TO "*)"  NESTED

TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .
  EOG         = "$" .

PRODUCTIONS
  EBNF
  = { Production } EOG           (. bool haserrors = !Successful();
                                    GParser->checkgraph(stderr, haserrors);
                                    if (haserrors) SemError(200); .) .

  Production
  =                              (. GP_GRAPH rhs;
                                    GP_PROD lhs;
                                    char name[100]; .)
    NonTerminal<name>            (. GParser->startproduction(name, lhs); .)
    "=" Expression<rhs>          (. if (Successful())
                                      GParser->completeproduction(lhs, rhs); .)
    "." .

  Expression<GP_GRAPH &first>
  =                              (. GP_GRAPH next; .)
    Term<first>
    { "|" Term<next>             (. GParser->linkterms(first, next); .)
    } .

  Term<GP_GRAPH &first>
  =                              (. GP_GRAPH next; .)
    ( Factor<first>
      { Factor<next>             (. GParser->linkfactors(first, next); .)
      }
      |                          (. GParser->epsnode(first); .)
    ) .

  Factor<GP_GRAPH &node>
  =                              (. char name[100]; .)
      NonTerminal<name>          (. GParser->nonterminalnode(name, node); .)
    | Terminal<name>             (. GParser->terminalnode(name, node); .)
    | "[" Expression<node> "]"   (. GParser->optionalnode(node); .)
    | "{" Expression<node> "}"   (. GParser->repeatednode(node); .)
    | "(" Expression<node> ")" .

  NonTerminal<char *name>
  =  nonterminal                 (. LexName(name, 100); .) .

  Terminal<char *name>
  =  terminal                    (. char local[100];
                                    LexName(local, sizeof(local) - 1);
                                    int i = 0; /* strip quotes */
                                    while (local[i])
                                    { local[i] = local[i+1]; i++; }
                                    local[i-2] = '\0';
                                    strcpy(name, local); .) .

END EBNF.
```

The simplicity here is deceptive: this system has delegated control to various node creation and linker routines that are members of an instance `GParser` of a general graph parser class `GP`. It is predominantly the task of *Factor* (at the lowest point in the hierarchy) to call the routines to generate new actual nodes in the graph: the task of the routines called from other functions is to link them correctly (that called from *Term* uses the `Next` field, while *Expression* uses the `Alternate` field).

A non-terminal might appear in an *Expression* before it has appeared on the left side of a production. In this case it is still entered into the list of rules by a call to the *StartProduction* routine.

Once one has constructed these sorts of structures, what can be done with them? The idea of a graph-walker can be used in various ways. In Coco/R such graph-walkers are used in conjunction with the frame files, merging appropriately generated source code with these files to produce complete programs.

Further exploration

An implementation of the `GP` class, and of an associated scanner class `GS` has been provided on the source diskette, and will allow the reader to study these ideas in more detail. Be warned that the code, while quite concise, is not particularly easy to follow - and is still a long way short of being a program that can handle attributes and perform checks that the grammar submitted to it satisfies constraints like LL(1) conditions. Furthermore, the code does not demonstrate the construction of a complete parser generator, although it does show the development of a simple direct graph driven parser based on that suggested by Wirth (1976b, 1996).

This is actually a very naïve parsing algorithm, requiring rather special constraints on the grammar. It has the property of pursuing a new subgoal whenever it appears (by virtue of the recursive call to `ParseFrom`), without first checking whether the current symbol is in the set FIRST(*Goal->RightSide*). This means that the syntax must have been described in a rather special way - if a *NonTerminal* is nullable, then none of its right parts must start with a non-terminal, and each *Factor* (except possibly the last one) in the group of alternatives permitted by a *Term* must start with a distinct terminal symbol.

So, although this parser sometimes appears to work quite well - for example, for the *ClassList* grammar above it will correctly report that input sentences like

```
CS3 BSc : Tom, Dick ,, Harry .
CS3 BScS : Tom Dick .
```

are malformed - it will accept erroneous input like

```
CS3 BSc : .
```

as being correct. The assiduous reader might like to puzzle out why this is so.

The source code for Coco/R, its support modules, and the attributed grammar from which it is bootstrapped, are available from various Internet sites, as detailed in Appendix A. The really curious reader is encouraged to obtain copies of these if he or she wishes to learn more about Coco/R itself, or about how it is used in the construction of really large applications.

13.5 Project suggestions

Coco/R, like other parser generators, is a very powerful tool. Here are some suggestions for further projects that the reader might be encouraged to undertake.

13.17 The various expression parsers that have been used in earlier case studies have all assumed that the operands are simple integers. Suppose we wished to extend the underlying grammar to allow for comparison operations (which would operate on integer values but produce Boolean results), arithmetic operations (which operate on integer values and produce integer results) and logical operations (which act on Boolean values to produce Boolean results). A context-free grammar for such expressions, based on that used in Pascal and Modula-2, is given below. Incorporate this into an attributed Cocol grammar that will allow you to check whether expressions are semantically acceptable (that is, whether the operators have been applied in the correct context). Some examples follow

```
Acceptable                Not acceptable

3 + 4 * 6                 3 + 4 < 6
(x > y) AND (a < b)       x < y OR a < b
```

```
Expression       =  SimpleExpression [ RelOp SimpleExpression ] .
SimpleExpression =  Term { AddOp Term } .
Term             =  Factor { MulOp Factor } .
Factor           =   identifier | number  | "(" Expression ")"
                    | "NOT" Factor | "TRUE" | "FALSE" .
AddOp            =  "+" | "-" | "OR" .
MulOp            =  "*" | "/" | "AND"  .
RelOp            =  "<" | "<=" | ">" | ">=" | "=" | "<>" .
```

13.18 The "spreadsheet" has become a very popular tool in recent years. This projects aims to use Coco/R to develop a simple spreadsheet package.

A modern commercial package provides many thousands of features; we shall be less ambitious. In essence a simple two-dimensional spreadsheet is based on the concept of a matrix of cells, typically identified by a letter-digit pair (such as `E7`) in which the letter specifies a row, and the digit specifies a column. Part (or all) of this matrix is displayed on the terminal screen; one cell is taken as the *active cell*, and is usually highlighted in some way (for example, in inverse video).

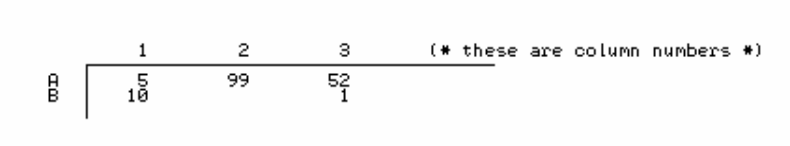
Input to a spreadsheet is then provided in the form of expressions typed by the user, interleaved with commands that can reselect the position of the active cell. Each time an expression is typed, its *formula* is associated with the active cell, and its *value* is displayed in the correct position. Changing the contents of one cell may affect the values of other cells. In a very simple spreadsheet implementation, each time one cell is assigned a new expression, the values of all the other cells are recomputed and redisplayed.

For this exercise assume that the expressions are confined to integer expressions of the sort exhaustively discussed in this text. The operands may be integer literals, or the designators of cells. No attempt need be made to handle string or character values.

A simple session with such a spreadsheet might be described as follows

```
(* we start in cell A1 *)
1 RIGHT             (* enter 1 in cell A1 and move on to cell A2 *)
99 RIGHT            (* enter 99 in cell A2 and move on to cell A3 *)
(A1 + A2) / 2  ENTER (* cell A3 contains the average of A1 and A2 *)
DOWN LEFT LEFT      (* move to cell B1 *)
2 * A1              (* cell B1 now contains twice the value of A1 *)
UP                  (* move back to cell A1 *)
5                   (* alter expression in A1 : A3 and B1 affected *)
GOTO B3             (* move to cell B3 *)
A3 % 3 ENTER        (* B3 contains remainder when A3 is divided by 3 *)
QUIT
```

At the point just before we quit, the grid displayed on the top left of the screen might display



It is possible to develop such a system using Coco/R in a number of ways, but it is suggested that you proceed as follows:

(a) Derive a context-free grammar that will describe the form of a session with the spreadsheet like that exemplified above.

(b) Enhance your grammar to provide the necessary attributes and actions to enable a complete system to be generated that will read and process a file of input and compute and display the spreadsheet, updating the display each time new expressions become associated with cells.

Make the following simplifying assumptions:

(a) A spreadsheet is normally run "interactively". However, Coco/R generates systems that most conveniently take their input from a disk file. If you want to work interactively you will need to modify the scanner frame file considerably.

(b) Assume that the spreadsheet has only 20 rows and 9 columns, extending from A1 through S9.

(c) Apart from accepting expressions typed in an obvious way, assume that the movement commands are input as `LEFT, RIGHT, UP, DOWN, HOME` and `GOTO Cell` as exemplified above. Assume that attempts to move too far in one direction either "wrap around" (so that a sequence like `GOTO A1  UP` results in cell `S1` becoming the active cell; `GOTO A12` actually moves to `A3`, and so on) or simply "stick" at the edge, as you please.

(d) An expression may also be terminated by `ENTER`, which does not affect the selection of the active cell.

(e) Input to the spreadsheet is terminated by the `QUIT` operation.

(f) The semantics of updating the spreadsheet display are captured in the following pseudo-code:

```
When Expression is recognized as complete
  Store Expression[CurrentRow, CurrentColumn] in a form
        that can be used for future interpretation
  Update value of Value[CurrentRow, CurrentColumn]
  FOR Row FROM A TO S DO
    FOR Column FROM 1 TO 9 DO
      Update Value[Row, Column] by
        evaluating Expression[Row, Column]
      Display new Value[Row, Column]
    END
  END
```

(g) Arrange that the spreadsheet starts with the values of each cell set to zero, and with no expressions associated with any cell.

(h) No facilities for "editing" an expression need be provided; if a cell’s expression is to be altered it must be typed afresh.

Hint: The most intriguing part of this exercise is deciding on a way to store an expression so that it can be evaluated again when needed. It is suggested that you associate a simple auxiliary data

structure with each cell of the spreadsheet. Each element of this structure can store an operation or operand for a simple interpreter.

13.19 A rather useful tool to have when dealing with large amounts of source code is a "cross reference generator". This is a program that will analyse the source text and produce a list of all the identifiers that appear in it, along with a list for each identifier of the line numbers on which it can be found. Construct a cross reference generator for programs written in Clang, for which a grammar was given in section 8.7, or for one of the variations on it suggested in Exercises 8.25 through 8.30. This can be done at various levels of sophistication; you should at least try to distinguish between the line on which an identifier is "declared", and those where it is "applied". A useful way to decompose the problem might be to develop a support module with an interface to a hidden data structure:

```
void Create();
// Initialize a new (empty) Table

void Add(char *Name, int Reference, bool Defining);
// Add Name to Table with given Reference, specifying whether
// this is a Defining (as opposed to an applied occurrence)

void List(FILE *lst);
// List out cross reference Table on lst file
```

You should then find that the actions needed to enhance the grammar are very straightforward, and the bulk of any programming effort falls on the development of a simple tree or queue-based data structure similar to those which you should have developed in other courses you have taken in Computer Science.

13.20 In case you have not met this concept before, a pretty printer is a "compiler" that takes a source program and "translates" the source into the same language. That probably does not sound very useful! However, the "object code" is formatted neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

Develop a pretty printer for the simple Clang language for which the grammar was given in section 8.7. The good news is that you will not have to develop any semantic analysers, code generators, or symbol table handlers in this project, but can assume that the source program is semantically correct if it is syntactically correct. The bad news is that you may have some difficulty in retaining the comments. They can no longer be ignored, but should preferably be copied across to the output in some way.

An obvious starting point is to enhance the grammar with actions that simply write output as terminals are parsed. An example will make this clearer

```
CompoundStatement =
  "BEGIN"               (. Append("BEGIN"); IndentNewLine(); .)
     Statement
       { ";"            (. Append(";"); NewLine(); .)
         Statement }
  "END"                 (. ExdentNewLine(); Append("END"); .)   .
```

Of course, the productions for all the variations on *Statement* append their appropriate text as they are unravelled.

Once again, an external module might conveniently be introduced to give the support needed for these semantic actions, perhaps with an interface on the lines of

```
void Append(char *String);
// Append String to output
```

```
void IndentNewLine(void);
// Write line mark to output, and then prepare to indent further
// lines by a fixed amount more than before

void ExdentNewLine(void);
// Write line mark to output, and then prepare to indent further
// lines by a fixed amount less than before

void NewLine(void);
// Write line mark to output, but leave indentation as before

void Indent(void);
// Increment indentation level

void Exdent(void);
// Decrement indentation level

void SetIndentationStep(int Step);
// Set indentation step size to Step
```

13.21 If two high level languages are very similar, a translator from one to the other can often be developed by taking the idea of a pretty printer one stage further - rather than writing the same terminals as it reads, it writes slightly different ones. For example, a Clang *CompoundStatement* would be translated to the equivalent Topsy version by attributing the production as follows:

```
CompoundStatement =
  "BEGIN"               (. Append("{"); IndentNewLine(); .)
     Statement
       { ";"            (. NewLine(); .)
         Statement }
  "END"                 (. ExdentNewLine(); Append("}"); .)   .
```

Develop a complete Clang - Topsy translator in this way.

13.22 The Computer Centre has decided to introduce a system of charging users for electronic mail messages. The scale of charges will be as follows:

- Message charge: 20 units plus a charge per word of message text: 10 units for each word with at most 8 characters, 60 units for each word with more than 8 characters.
- The total charge is applied to every copy of the message transmitted - if a message is addressed to *N* multiple users, the sender's account is debited by *N* * *Charge*.

The program will be required to process data files exemplified by the following (read the messages - they give you some hints):

```
From: cspt@cs.ru.ac.za
To:   reader@in.bed, guru@sys-admin.uni-rhodes.ac.za
CC:   cslect@cs, pdterry@psg.com
This is a message containing twenty-seven words
The charge will be 20 plus 24 times 10 plus 3 times 60 units -
total 440 multiplied by 4
####
From: tutor@cs
To:   students@lab.somewhere
You should note that messages contain only words composed of plain
text or numbers or possible - signs

Assume for this project that no punctuation marks or other extra
characters will ever appear - this will make it much easier to do

User names and addresses may also contain digits and - characters
####
```

Each message has mandatory "`From`" and "`To`" lines, and an optional "`CC`" (carbon copy) line. Users are addressed in the usual Internet form, and case is insignificant. Ends of lines are, however, significant in addressing, and hence an EOL token must be catered for.

The chargeable text of a message starts after the `To` or `CC` line, and is terminated by the

(non-chargeable) #### line.

Describe this input by means of a suitable grammar, and then enhance it to provide the necessary attributes and actions to construct a complete charging system that will read and process a file of messages and then list the charges. In doing so you might like to consider developing a support module with an interface on the lines of that suggested below, and you should take care to incorporate error recovery.

```
void ChargeUser(char *Sender; int Charge);
// Pre:  Sender contains unique user name extracted from a From line
//       For example  cspt  extracted from  From: cspt@somewhere.com
//       Charge contains the charge for sending all copies of message
// Post: Database of charges updated to debit Charge to Sender

void ShowCharges(FILE *F);
// Pre:  Opened(F) AND the internal data base contains a list of user
//       names and accrued charges
// Post: The list has been displayed on file F
```

13.23 (This project requires some familiarity with music). "Tonic Solfa" is a notation sometimes used to help learn to play an instrument, or more frequently to sing, without requiring the use of expensive music printed in "staff notation". Many readers may have come across this as it applies to representing pitch. The notes of a major scale are named *doh, ray, me, fah, soh, lah, te* (and, as Julie Andrews taught us in *The Sound of Music*, that brings us back to *doh*). In the written notation these syllables are indicated by their initial letters only: *d r m f s l t*. Sharpened notes are indicated by adding the letter *e*, and flattened notes by adding the letter *a* (so that if the major scale were C major, *fe* would indicate F sharp and *la* would indicate A flat). Notes in octaves above the "starting" *doh* are indicated by superscript numbers, and notes below the "starting" *doh* are indicated by subscripts. Although the system is basically designed to indicate relative pitch, specific keys can be named at the beginning of the piece.

If, for the moment, we ignore timing information, the notes of the well-known jingle "Happy Birthday To You" could be represented by

$s_1\ s_1\ l_1\ s_1\ d\ t_1\ s_1\ s_1\ l_1\ s_1\ r\ d$
$s_1\ s_1\ s\ m\ d\ t_1\ l_1\ f\ f\ m\ d\ r\ d$

Of course we cannot really ignore timing information, which, unfortunately, complicates the picture considerably. In this notation, bar lines | and double bar lines || appear much as in staff notation. Braces { and } are used at the beginning and end of every line (except where a double bar line occurs).

The notation indicates relative note lengths, according to the basic pulse of the music. A bar line is placed before a strong pulse, a colon is placed before a weak pulse, and a shorter vertical line | indicates the secondary accent at the half bar in quadruple time. Horizontal lines indicate notes lasting longer than one beat (including dotted or tied notes). Pulses are divided in half by using dots as separators, and half pulses are further divided into quarter pulses by commas. Rests are indicated simply by leaving spaces. For example

| d : d | indicates duple time with notes on each pulse (two crotchets, if it were 2/4 time)

| d : - | d : d | indicates quadruple time (minim followed by two crotchets, in 4/4 time)

| d : - . d : | indicates triple time (dotted crotchet, quaver, crotchet rest, in 3/4 time)

| d : d . d : d,d . d | indicates triple time (crotchet, two quavers, two semiquavers, quaver, in 3/4
time)

"Happy Birthday To You" might then be coded fully as

{| : : s_1 . - , s_1 | l_1 : s_1 : d | t_1 : - : s_1 . - , s_1 }
{| l_1 : s_1 : r | d : - : s_1 . - , s_1 | s : m : d }
{| t_1 : l_1 : f . - , f | m : d : r | d : - : ||

Clearly this is fairly complex, and one suspects that singers may learn the rhythms "by ear" rather than by decoding this as they sing!

Write a Cocol grammar that describes this notation. Then go on to use it to develop a program that can read in a tune expressed this way and produce "machine code" for a device that is driven by a long stream of pairs of numbers, the first indicating the frequency of the note in Hz, and the second the duration of that note in milliseconds.

Recognizing superscripts and subscripts is clearly awkward, and it is suggested that you might initially use *d0 r0 m0 f0 s0 l0 t0 d r m f s l t d1 r1 m1 f1 s1 l1 t1* to give a range of three octaves, which will suffice for most songs.

Initially you might like to assume that a time signature (like the key) will preface the piece (which will simplify the computation of the duration of each note), and that the timing information has been correctly transcribed. As a later extension you might like to consider how varying time signatures and errors in transcription could be handled, while still assuming that each bar takes the same time to play.

14 A SIMPLE COMPILER - THE FRONT END

At this point it may be of interest to consider the construction of a compiler for a simple programming language, specifically that of section 8.7. In a text of this nature it is impossible to discuss a full-blown compiler, and the value of our treatment may arguably be reduced by the fact that in dealing with toy languages and toy compilers we shall be evading some of the real issues that a compiler writer has to face. However, we hope the reader will find the ensuing discussion of interest, and that it will serve as a useful preparation for the study of much larger compilers. The technique we shall follow is one of slow refinement, supplementing the discussion with numerous asides on the issues that would be raised in compiling larger languages. Clearly, we could opt to develop a completely hand-crafted compiler, or simply to use a tool like Coco/R. We shall discuss both approaches. Even when a compiler is constructed by hand, having an attributed grammar to describe it is very worthwhile.

On the source diskette can be found a great deal of code, illustrating different stages of development of our system. Although some of this code is listed in appendices, its volume precludes printing all of it. Some of it has deliberately been written in a way that allows for simple modification when attempting the exercises, and is thus not really of "production quality". For example, in order to allow components such as the symbol table handler and code generator to be used either with hand-crafted or with Coco/R generated systems, some compromises in design have been necessary.

Nevertheless, the reader is urged to study the code along with the text, and to attempt at least some of the many exercises based on it. A particularly worthwhile project is to construct a similar compiler, based on a language whose syntax resembles C++ rather more than it does Pascal, and whose development, like that of C++, will be marked by the steady assimilation of extra features. This language we shall name "Topsy", after the little girl in Harriet Beecher Stowe’s story who knew little of her genealogy except a suspicion that she had "grow’d". A simple Topsy program was illustrated in Exercise 8.25, where the reader was invited to create an initial syntactic specification in Cocol.

14.1 Overall compiler structure

In Chapter 2 we commented that a compiler is often developed as a sequence of phases, of which syntactic analysis is only one. Although a recursive descent parser is easily written by applying the ideas of earlier chapters, it should be clear that consideration will have to be given to the relationship of this to the other phases. We can think of a compiler with a recursive descent parser at its core as having the structure depicted in Figure 14.1.

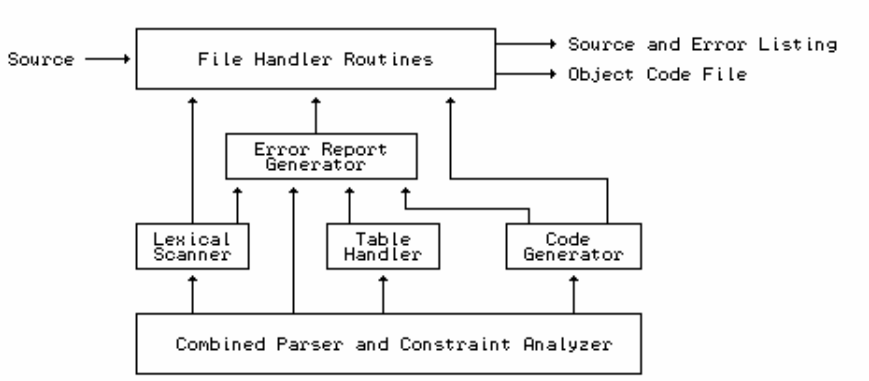


Figure 14.1 Relationship between the main components of a simple compiler

We emphasize that phases need not be sequential, as passes would be. In a recursive descent compiler the phases of syntax analysis, semantic analysis and code generation are very often interleaved, especially if the source language is designed in such a way as to permit one-pass compilation. Nevertheless, it is useful to think of developing modular components to handle the various phases, with clear simple interfaces between them.

In our Modula-2 implementations, the various components of our diagram have been implemented as separate modules, whose `DEFINITION MODULE` components export only those facilities that the clients need be aware of. The corresponding C++ implementations use classes to achieve the same sort of abstraction and protection.

In principle, the main routine of our compiler must resemble something like the following

```
void main(int argc, char *argv[])
{ char SourceName[256], ListName[256];

  // handle command line parameters
  strcpy(SourceName, argv[1]);
  if (argc > 2) strcpy(ListName, argv[2]);
  else appendextension(SourceName, ".lst", ListName);

  // instantiate compiler components
  SRCE   *Source  = new SRCE(SourceName, ListName, "Compiler Version 1", true);
  REPORT *Report  = new REPORT(Source);
  SCAN   *Scanner = new SCAN(Source, Report);
  CGEN   *CGen    = new CGEN(Report);
  TABLE  *Table   = new TABLE(Report);
  PARSER *Parser  = new PARSER(CGen, Scanner, Table, Report);

  // start compilation
  Parser->parse();
}
```

where we notice that instances of the various classes are constructed dynamically, and that their constructors establish links between them that correspond to those shown in Figure 14.1.

In practice our compilers do not look exactly like this. For example, Coco/R generates only a scanner, a parser, a rudimentary error report generator and a driver routine. The scanner and the source handling section of the file handler are combined into one module, and the routines for producing the error listing are generated along with the main driver module. The C++ version of Coco/R makes use of a standard class hierarchy involving parser, scanner and error reporter classes, and establishes links between the various instances of these classes as they are constructed. This gives the flexibility of having multiple instances of parsers or scanners within one system (however, our case studies will not exploit this power).

14.2 Source handling

Among the file handling routines is to be found one that has the task of transmitting the source, character by character, to the scanner or lexical analyser (which assembles it into symbols for subsequent parsing by the syntax analyser). Ideally this source handler should have to scan the program text only once, from start to finish, and in a one-pass compiler this should always be possible.

14.2.1 A hand-crafted source handler

The interface needed between source handler and lexical analyser is straightforward, and can be supplied by a routine that simply extracts the next character from the source each time it is invoked. It is convenient to package this with the routines that assume responsibility for producing a source listing, and, where necessary, producing an error message listing, because all these requirements will be input/output device dependent. It is useful to add some extra functionality, and so the public interface to our source handling class is defined by

```
class SRCE {
  public:
    FILE *lst;                  // listing file
    char ch;                    // latest character read

    void nextch(void);
    // Returns ch as the next character on this source line, reading a new
    // line where necessary.  ch is returned as NUL if src is exhausted.

    bool endline(void);
    // Returns true when end of current line has been reached

    void listingon(void);
    // Requests source to be listed as it is read

    void listingoff(void);
    // Requests source not to be listed as it is read

    void reporterror(int errorcode);
    // Points out error identified by errorcode with suitable message

    virtual void startnewline()  {;}
    // Called at start of each line

    int getline(void);
    // Returns current line number

    SRCE(char *sourcename, char *listname, char *version, bool listwanted);
    // Opens src and lst files using given names.
    // Resets internal state in readiness for starting to scan.
    // Notes whether listwanted.  Displays version information on lst file.

    ~SRCE();
    // Closes src and lst files
};
```

Some aspects of this interface deserve further comment:

- We have not shown the private members of the class, but of course there are several of these.
- The `startnewline` routine has been declared virtual so that a simple class can be derived from this one to allow for the addition of extra material at the start of each new line on the listing - for example, line numbers or object code addresses. In Modula-2, Pascal or C, the same sort of functionality may be obtained by manipulating a procedure variable or function pointer.
- Ideally, both source and listing files should remain private. This source handler declares the

listing file public only so that we can add trace or debugging information while the system is being developed.

- The class constructor and destructor assume responsibility for opening and closing the files, whose names are passed as arguments.

The implementation of this class is fairly straightforward, and has much in common with the similar class used for the assemblers of Chapter 6. The code appears in Appendix B, and the following implementation features are worthy of brief comment:

- The source is scanned and reflected a whole line at a time, as this makes subsequent error reporting much easier.

- The handler effectively inserts an extra blank at the end of each line. This decouples the rest of the system from the vagaries of whatever method the host operating system uses to represent line ends in text files. It also ensures that no symbol may extend over a line break.

- It is not possible to read past the end of file - attempts to do so simply return a `NUL` character.

- The `reporterror` routine will not display an error message unless a minimum number of characters have been scanned since the last error was reported. This helps suppress the cascade of error messages that might otherwise appear at any one point during error recovery of the sort discussed in sections 10.3 and 14.6.

- Our implementation has chosen to use the `stdio` library, rather than `iostreams`, mainly to take advantage of the concise facilities provided by the `printf` routine.

Exercises

14.1 The `nextch` routine will be called once for every character in the source text. This can represent a considerable bottleneck, especially as programs are often prepared with a great many blanks at the starts of indented lines. Some editors also pad out the ends of lines with unnecessary blanks. Can you think of any way in which this overhead might be reduced?

14.2 Some systems allowing an ASCII character set (with ordinal values in the range 0 ... 127) are used with input devices which generate characters having ordinal values in the range 0 ... 255 - typically the "eighth bit" might always be set, or used or confused with parity checking. How and where could this bit be discarded?

14.3 A source handler might improve on efficiency rather dramatically were it able to read the entire source file into a large memory buffer. Since modern systems are often blessed with relatively huge amounts of RAM, this is usually quite feasible. Develop such a source handler, compatible with the class interface suggested above, bearing in mind that we wish to be able to reflect the source line by line as it is read, so that error messages can be appended as exemplified in section 14.6.

14.4 Develop a source handler implementation that uses the C++ stream-based facilities from the `iostreams` library.

14.2.2 Source handling in Coco/R generated systems

As we have already mentioned, Coco/R integrates the functions of source handler and scanner, so as to be able to cut down on the number of files it has to generate. The source and listing files have to be opened before making the call to instantiate or initialize the scanner, but this is handled automatically by the generated driver routine. It is of interest that the standard frame files supplied with Coco/R arrange for this initialization to read the entire source file into a buffer, as suggested in Exercise 14.3.

14.3 Error reporting

As can be seen from Figure 14.1, most components of a compiler have to be prepared to signal that something has gone awry in the compilation process. To allow all of this to take place in a uniform way, we have chosen to introduce a base class with a very small interface:

```
class REPORT {
  public:
    REPORT();
    // Initializes error reporter

    virtual void error(int errorcode);
    // Reports on error designated by suitable errorcode number

    bool anyerrors(void);
    // Returns true if any errors have been reported

  protected:
    bool errors;
};
```

Error reporting is then standardized by calling on the `error` member of this class whenever an error is detected, passing it a unique number to distinguish the error.

The base class can choose simply to abort compilation altogether. Although at least one highly successful microcomputer Pascal compiler uses this strategy (Turbo Pascal, from Borland International), it tends to become very annoying when one is developing large systems. Since the `error` member is virtual, it is an easy matter to derive a more suitable class from this one, without, of course, having to amend any other part of the system. For our hand-crafted system we can do this as follows:

```
class clangReport : public REPORT {
  public:
    clangReport(SRCE *S)   { Srce = S; }
    virtual void error(int errorcode)
      { Srce->reporterror(errorcode); errors = true; }
  private:
    SRCE *Srce;
};
```

and the same technique can be used to enhance Coco/R generated systems. The Modula-2 and Pascal implementations achieve the same functionality through the use of procedure variables.

14.4 Lexical analysis

The main task of the scanner is to provide some way of uniquely identifying each successive token or symbol in the source code that is being compiled. Lexical analysis was discussed in section 10.4,

and presents few problems for a language as simple as ours.

14.4.1 A hand-crafted scanner

The interface between the scanner and the main parser is conveniently provided by a routine `getsym` for returning a parameter `SYM` of a record or structure type assembled from the source text. This can be achieved by defining a class with a public interface as follows:

```
enum SCAN_symtypes {
  SCAN_unknown, SCAN_becomes, SCAN_lbracket, SCAN_times, SCAN_slash, SCAN_plus,
  SCAN_minus, SCAN_eqlsym, SCAN_neqsym, SCAN_lsssym, SCAN_leqsym, SCAN_gtrsym,
  SCAN_geqsym, SCAN_thensym, SCAN_dosym, SCAN_rbracket, SCAN_rparen, SCAN_comma,
  SCAN_lparen, SCAN_number, SCAN_stringsym, SCAN_identifier, SCAN_coendsym,
  SCAN_endsym, SCAN_ifsym, SCAN_whilesym, SCAN_stacksym, SCAN_readsym,
  SCAN_writesym, SCAN_returnsym, SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym,
  SCAN_semicolon, SCAN_beginsym, SCAN_constsym, SCAN_varsym, SCAN_procsym,
  SCAN_funcsym, SCAN_period, SCAN_progsym, SCAN_eofsym
};

const int lexlength = 128;
typedef char lexeme[lexlength + 1];

struct SCAN_symbols {
  SCAN_symtypes sym;    // symbol type
  int num;              // value
  lexeme name;          // lexeme
};

class SCAN {
  public:
    void getsym(SCAN_symbols &SYM);
    // Obtains the next symbol in the source text

    SCAN(SRCE *S, REPORT *R);
    // Initializes scanner
};
```

Some aspects of this interface deserve further comment:

- `SCAN_symbols` makes provision for returning not only a unique symbol type, but also the corresponding textual representation (known as a **lexeme**), and also the numeric value when a symbol is recognized as a number.

- `SCAN_unknown` caters for erroneous characters like # and ? which do not really form part of the terminal alphabet. Rather than take action, the humble scanner returns the symbol without comment, and leaves the parser to cope. Similarly, an explicit `SCAN_eofsym` is always returned if `getsym` is called after the source code has been exhausted.

- The ordering of the `SCAN_symtypes` enumeration is significant, and supports an interesting form of error recovery that will be discussed in section 14.6.1.

- The enumeration has also made provision for a few symbol types that will be used in the extensions of later chapters.

A scanner for Clang is readily programmed in an *ad hoc* manner, driven by a selection statement, and an implementation can be found in Appendix B. As with source handling, some implementation issues call for comment:

- Some ingenuity has to be applied to the recognition of literal strings. A repeated quote within a string is used (as in Pascal) to denote a single quote, so that the end of a string can only be detected when an odd number of quotes is followed by a non-quote.

- The scanner has assumed responsibility for a small amount of semantic activity, namely the evaluation of a number. Although it may seem a convenient place to do this, such analysis is not always as easy as it might appear. It becomes slightly more difficult to develop scanners that have to distinguish between numbers represented in different bases, or between real and integer numbers.

- There are two areas where the scanner has been given responsibility for detecting errors:

  Although the syntactic description of the language does not demand it, practical considerations require that the value of a numeric constant should be within the range of the machine. This is somewhat tricky to ensure in the case of cross-compilers, where the range on the host and target machines may be different. Some authors go so far as to suggest that this semantic activity be divorced from lexical analysis for that reason. Our implementation shows how range checking can be handled for a self-resident compiler.

  Not only do many languages insist that no identifier (or any other symbol) be carried across a line break, they usually do this for strings as well. This helps to guard against the chaos that would arise were a closing quote to be omitted - further code would become string text, and future string text would become code! The limitation that a string be confined to one source line is, in practice, rarely a handicap, and the restriction is easily enforced.

- We have chosen to use a binary search to recognize the reserved keywords. Tables of symbol types that correspond to keywords, and symbols that correspond to single character terminals, are initialized as the scanner is instantiated. The idioms of C++ programming suggest that such activities are best achieved by having static members of the class, set up by a "initializer" that forms part of their definition. For a binary search to function correctly it is necessary that the table of keywords be in alphabetic order, and care must be taken if and when the scanner is extended.

Exercises

14.5 The only screening this scanner does is to strip blanks separating symbols. How would you arrange for it to strip comments

(a) of the form `{ comment in curly braces }`
(b) of the form `(* comment in Modula-2 braces *)`
(c) of the form `// comment to end of the line as in C++`
(d) of either or both of forms (a) and (b), allowing for nesting?

14.6 Balanced comments are actually dangerous. If not properly closed, they may consume valid source code. One way of assisting the coder is to issue a warning if a semicolon is found within a comment. How could this be implemented as part of the answer to Exercise 14.5?

14.7 The scanner does not react sensibly to the presence of tab or formfeed characters in the source. How can this be improved?

14.8 Although the problem does not arise in the case of Clang, how do you suppose a hand-crafted scanner is written for languages like Modula-2 and Pascal that must distinguish between `REAL` literals of the form `3.4` and subrange specifiers of the form `3..4`, where no spaces delimit the "..",

as is quite legal? Can you think of an alternative syntax which avoids the issue altogether? Why do you suppose Modula-2 and Pascal do not use such a syntax?

14.9 Modula-2 allow string literals to be delimited by either single or double quotes, but not to contain the delimiter as a member of the string. C and C++ use single and double quotes to distinguish between character literals and string literals. Develop scanners that meet such requirements.

14.10 In C++, two strings that appear in source with nothing but white space between them are automatically concatenated into a single string. This allows long strings to be spread over several lines, if necessary. Extend your scanner to support this feature.

14.11 Extend the scanner to allow escape sequences like the familiar `\n` (newline) or `\t` (tab) to represent "control" characters in literal strings, as in C++.

14.12 Literal strings present other difficulties to the rest of the system that may have to process them. Unlike identifiers (which usually find their way into a symbol table), strings may have to be stored in some other way until the code generator is able to handle them. Consider extending the `SCAN_symbols` structure so that it contains a member that points to a dynamically allocated array of exactly the correct length for storing any string that has been recognized (and is a null pointer otherwise).

14.13 In our compiler, a diagnostic listing of the symbol table will be provided if the name `Debug` is used for the main program. Several compilers make use of pragmatic comments as compiler directives, so as to make such demands of the system - for example a comment of the form `(*$L- *)` might request that the listing be switched off, and one of the form `(*$L+ *)` that it be reinstated. These requests are usually handled by the scanner. Implement such facilities for controlling listing of the source program, and listing the symbol table (for example, using `(*$T+ *)` to request a symbol table listing). What action should be taken if a source listing has been suppressed, and if errors are discovered?

14.14 The restriction imposed on the recognizable length of a lexeme, while generous, could prove embarrassing at some stage. If, as suggested in Exercise 14.3, a source handler is developed that stores the entire source text in a memory buffer, it becomes possible to use a less restrictive structure for `SCAN_symbols`, like that defined by

```
struct SCAN_symbols {
  SCAN_symtypes sym;  // symbol type
  int num;            // value
  long pos, length;   // starting position and length of lexeme
};
```

Develop a scanner based on this idea. While this is easy to do, it may have ramifications on other parts of the system. Can you predict what these might be?

14.15 Develop a hand-crafted scanner for the Topsy language of Exercise 8.25. Incorporate some of the features suggested in Exercises 14.5 to 14.14.

14.4.2 A Coco/R generated scanner

A Cocol specification of the token grammar for our language is straightforward, and little more need be said. In C++, the generated scanner class is derived from a standard base class that assumes that the source file has already been opened; its constructor takes an argument specifying the corresponding "file handle". As we have already noted in Chapter 12, calls to the `Get` routine of this

scanner simply return a token number. If we need to determine the text of a string, the name of an identifier, or the value of a numeric literal, we are obliged to write appropriately attributed productions into the phrase structure grammar. This is easily done, as will be seen by studying these productions in the grammars to be presented later.

Exercises

14.16 Is it possible to write a Cocol specification that generates a scanner that can handle the suggestions made in Exercises 14.10 and 14.11 (allowing strings that immediately follow one another to be automatically concatenated, and allowing for escape sequences like "\n" to appear within strings to have the meanings that they do in C++)? If not, how else might such features be incorporated into Coco/R generated systems?

14.4.3 Efficient keyword recognition

The subject of keyword recognition is important enough to warrant further comment. It is possible to write a FSA to do this directly (see section 10.5). However, in most languages, including Clang and Topsy, identifiers and keywords have the same basic format, suggesting the construction of scanners that simply extract a "word" into a string, which is then tested to see whether it is, in fact, a keyword. Since string comparisons are tedious, and since typically 50%-70% of program text consists of either identifiers or keywords, it makes sense to be able to perform this test as quickly as possible. The technique used in our hand-crafted scanner of arranging the keywords in an alphabetically ordered table and then using a binary search is only one of several ideas that strive for efficiency. At least three other methods are often advocated:

- The keywords can be stored in a table in length order, and a sequential search used among those that have the same length as the word just assembled.
- The keywords can be stored in alphabetic order, and a sequential search used among those that have the same initial letter as the word just assembled. This is the technique employed by Coco/R.
- A "perfect hashing function" can be derived for the keyword set, allowing for a single string comparison to distinguish between all identifiers and keywords.

A hashing function is one that is applied to a string so as to extract particular characters, map these onto small integer values, and return some combination of those. The function is usually kept very simple, so that no time is wasted in its computation. A *perfect* hash function is one chosen to be clever enough so that its application to each of the strings in a set of keywords results in a unique value for each keyword. Several such functions are known. For example, if we use an ASCII character set, then the C++ function

```
int hash (char *s)
{ int L = strlen(s); return (256 * s[0] + s[L-1] + L) % 139; }
```

will return 40 unique values in the range 0 ... 138 when applied to the 40 strings that are the keywords of Modula-2 (Gough and Mohay, 1988). Of course, it will return some of these values for non-keywords as well (for example the keyword "VAR" maps to the value 0, as does any other three letter word starting with "V" and ending with "R"). To use this function one would first construct a 139 element string table, with the appropriate 40 elements initialized to store the keywords, and the

rest to store null strings. As each potential identifier is scanned, its hash value is computed using the above formula. A single probe into the table will then ascertain whether the word just recognized is a keyword or not.

Considerable effort has gone into the determination of "minimal perfect hashing functions" - ones in which the number of possible values that the function can return is exactly the same as the number of keywords. These have the advantage that the lookup table can be kept small (Gough's function would require a table in which nearly 60% of the space was wasted).

For example, when applied to the 19 keywords used for Clang, the C++ function

```
int hash (char *s)
{ int L = strlen(s); return Map[s[0]] + Map[s[L-2]] + L - 2; }
```

will return a unique value in the range 0 ... 18 for each of them. Here the mapping is done via a 256 element array `Map`, which is initialized so that all values contain zero save for those shown below:

```
Map['B'] = 6; Map['D'] = 8; Map['E'] =  5; Map['L'] = 9;
Map['M'] = 7; Map['N'] = 8; Map['O'] = 12; Map['P'] = 3;
Map['S'] = 3; Map['T'] = 8; Map['W'] =  1;
```

Clearly this particular function cannot be applied to strings consisting of a single character, but such strings can easily be recognized as identifiers anyway. It is one of a whole class of similar functions proposed by Cichelli (1980), who also developed a backtracking search technique for determining the values of the elements of the `Map` array.

It must be emphasized that if a perfect hash function technique is used for constructing scanners for languages like Clang and Topsy that are in a constant state of flux as new keywords are proposed, then the hash function has to be devised afresh with each language change. This makes it an awkward technique to use for prototyping. However, for production quality compilers for well established languages, the effort spent in finding a perfect hash function can have a marked influence on the compilation time of tens of thousands of programs thereafter.

Exercises

To assist with these exercises, a program incorporating Cichelli's algorithm, based on the one published by him in 1979, appears on the source diskette. Another well known program for the construction of perfect hash functions is known as `gperf`. This is written in C, and is available from various Internet sites that mirror the extensive GNU archives of software distributed by the Free Software Foundation (see Appendix A).

14.17 Develop hand-crafted scanners that make use of the alternative methods of keyword identification suggested here.

14.18 Carry out experiments to discover which method seems to be best for the reserved word lists of languages like Clang, Topsy, Pascal, Modula-2 or C++. To do so it is not necessary to develop a full scale parser for each of these languages. It will suffice to invoke the `getsym` routine repeatedly on a large source program until all symbols have been scanned, and to time how long this takes.

Further reading

Several texts treat lexical analysis in far more detail than we have done; justifiably, since for larger languages there are considerably more problem areas than our simple one raises. Good discussions are found in the books by Gough (1988), Aho, Sethi and Ullman (1986), Welsh and Hay (1986) and Elder (1994). Pemberton and Daniels (1982) give a very detailed discussion of the lexical analyser found in the Pascal-P compiler.

Discussion of perfect hash function techniques is the source of a steady stream of literature. Besides the papers by Cichelli (1979, 1980), the reader might like to consult those by Cormack, Horspool and Kaiserwerth (1985), Sebesta and Taylor (1985), Panti and Valenti (1992), and Trono (1995).

14.5 Syntax analysis

For languages like Clang or Topsy, which are essentially described by LL(1) grammars, construction of a simple parser presents few problems, and follows the ideas developed in earlier chapters.

14.5.1 A hand-crafted parser

Once again, if C++ is the host language, it is convenient to define a hand-crafted parser in terms of its own class. If all that is required is syntactic analysis, the public interface to this can be kept very simple:

```
class PARSER {
  public:
    PARSER(SCAN *S, REPORT *R);
    // Initializes parser

    void parse(void);
    // Parses the source code
};
```

where we note that the class constructor associates the parser instance with the appropriate instances of a scanner and error reporter. Our complete compiler will need to go further than this - an association will have to be made with at least a code generator and symbol table handler. As should be clear from Figure 14.1, in principle no direct association need be made with a source handler (in fact, our system makes such an association, but only so that the parser can direct diagnostic output to the source listing).

An implementation of this parser, devoid of any attempt at providing error recovery, constraint analysis or code generation, is provided on the source diskette. The reader who wishes to see a much larger application of the methods discussed in section 10.2 might like to study this. In this connection it should be noted that Modula-2 and Pascal allow for procedures and functions to be nested. This facility (which is lacking in C and C++) can be used to good effect when developing compilers in those languages, so as to mirror the highly embedded nature of the phrase structure grammar.

14.5.2 A Coco/R generated parser

A parser for Clang can be generated immediately from the Cocol grammar presented in section 8.7.2. At this stage, of course, no attempt has been made to attribute the grammar to incorporate error recovery, constraint analysis, or code generation.

Exercises

Notwithstanding the fact that the construction of an parser that does little more than check syntax is still some distance away from having a complete compiler, the reader might like to turn his or her attention to some of the following exercises, which suggest extensions to Clang or Topsy, and to construct grammars, scanners and parsers for recognizing such extensions.

14.19 Compare the hand-crafted parser found on the source diskette with the source code that is produced by Coco/R.

14.20 Develop a hand-crafted parser for Topsy as suggested by Exercise 8.25.

14.21 Extend your parser for Clang to accept the `REPEAT ... UNTIL` loop as it is found in Pascal or Modula-2, or add an equivalent `do` loop to Topsy.

14.22 Extend the `IF ... THEN` statement to provide an `ELSE` clause.

14.23 How would you parse a Pascal-like `CASE` statement? The standard Pascal `CASE` statement does not have an `ELSE` or `OTHERWISE` option. Suggest how this could be added to Clang, and modify the parser accordingly. Is it a good idea to use `OTHERWISE` or `ELSE` for this purpose - assuming that you already have an `IF ... THEN ... ELSE` construct?

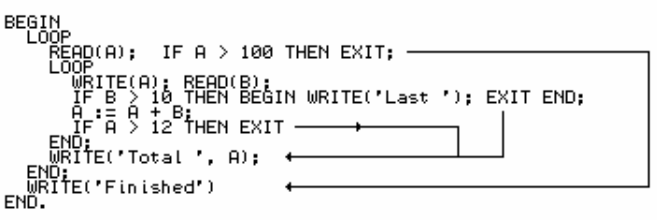
14.24 What advantages does the Modula-2 `CASE` statement have over the Pascal version? How would you parse the Modula-2 version?

14.25 The C++ `switch` statement bears some resemblance to the `CASE` statement, although its semantics are rather different. Add the `switch` statement to Topsy.

14.26 How would you add a Modula-2 or Pascal-like `FOR` loop to Clang?

14.27 The C++ `for` statement is rather different from the Pascal one, although it is often used in much the same way. Add a `for` statement to Topsy.

14.28 The `WHILE, FOR` and `REPEAT` loops used in Wirth's languages are *structured* - they have only one entry point, and only one exit point. Some languages allow a slightly less structured loop, which has only one entry point, but which allows exit from various places within the loop body. An example of this might be as follows



Like others, `LOOP` statements can be nested. However, `EXIT` statements may only appear within `LOOP` sequences. Can you find context-free productions that will allow you to incorporate these statements into Clang?

14.29 If you are extending Topsy to make it resemble C++ as closely as possible, the equivalent of the `EXIT` statement would be found in the `break` or `continue` statements that C++ allows within its various structured statements like `switch`, `do` and `while`. How would you extend the grammar for Topsy to incorporate these statements? Can the restrictions on their placement be expressed in a context-free grammar?

14.30 As a more challenging exercise, suppose we wished to extend Clang or Topsy to allow for variables and expressions of other types besides integer (for example, Boolean). Various approaches might be taken, as exemplified by the following

(a) Replacing the Clang keyword `VAR` by a set of keywords used to introduce variable lists:

```
int  X, Y, Z[4];
bool InTime, Finished;
```

(b) Retention of the `VAR` symbol, along with a set of standard type identifiers, used after variable lists, as in Pascal or Modula-2:

```
VAR
  X, Y, Z[4] : INTEGER;
  InTime, Finished : BOOLEAN;
```

Develop a grammar (and parser) for an extended version of Clang or Topsy that uses one or other of these approaches. The language should allow expressions to use Boolean operators (`AND, OR, NOT`) and Boolean constants (`TRUE` and `FALSE`). Some suggestions were made in this regard in Exercise 13.17.

14.31 The approach used in Pascal and Modula-2 has the advantage that it extends seamlessly to the more general situations in which users may introduce their own type identifiers. In C++ one finds a hybrid: variable lists may be preceded either by special keywords or by user defined type names:

```
typedef bool sieve[1000];
int X, Y;      // introduced by keyword
sieve Primes;  // introduced by identifier
```

Critically examine these alternative approaches, and list the advantages and disadvantages either seems to offer. Can you find context-free productions for Topsy that would allow for the introduction of a simple `typedef` construct?

A cynic might contend that if a language has features which are the cause of numerous beginners' errors, then one should redesign the language. Consider a selection of the following:

14.32 Bailes (1984) made a plea for the introduction of a "Rational Pascal". According to him, the keywords `DO` (in `WHILE` and `FOR` statements), `THEN` (in `IF` statements) and the semicolons which are used as terminators at the ends of declarations and as statement separators should all be discarded. (He had a few other ideas, some even more contentious). Can you excise semicolons from Clang and Topsy, and then write a recursive descent parser for them? If, indeed, semicolons seem to serve no purpose other than to confuse learner programmers, why do you suppose language designers use them?

14.33 The problems with `IF ... THEN` and `IF ... THEN ... ELSE` statements are such that one might be tempted to try a language construct described by

```
IfStatement =  "IF" Condition "THEN" Statement
                  { "ELSIF" Condition "THEN" Statement }
                  [ "ELSE Statement ] .
```

Discuss whether this statement form might easily be handled by extensions to your parser. Does it have any advantages over the standard `IF ... THEN ... ELSE` arrangement - in particular, does it resolve the "dangling else" problem neatly?

14.34 Extend your parser to accept structured statements on the lines of those used in Modula-2, for example

```
IfStatement        =  "IF" Condition "THEN" StatementSequence
                         { "ELSIF" Condition "THEN" StatementSequence }
                         [ "ELSE" StatementSequence  ]
                      "END" .
WhileStatement     =  "WHILE" Condition "DO" StatementSequence  "END" .
StatementSequence  =  Statement { ";" Statement } .
```

14.35 Brinch Hansen (1983) did not approve of implicit "empty" statements. How do these appear in our languages, are they ever of practical use, and if so, in what ways would an explicit statement (like the `SKIP` suggested by Brinch Hansen) be any improvement?

14.36 Brinch Hansen incorporated only one form of loop into Edison - the `WHILE` loop - arguing that the other forms of loops were unnecessary. What particular advantages and disadvantages do these loops have from the points of view of a compiler writer and a compiler user respectively? If you were limited to only one form of loop, which would you choose, and why?

14.6 Error handling and constraint analysis

In section 10.3 we discussed techniques for ensuring that a recursive descent parser can recover after detecting a syntax error in the source code presented to it. In this section we discuss how best to apply these techniques to our Clang compiler, and then go on to discuss how the parser can be extended to perform context-sensitive or constraint analysis.

14.6.1 Syntax error handling in hand-crafted parsers

The scheme discussed previously - in which each parsing routine is passed a set of "follower" symbols that it can use in conjunction with its own known set of "first" symbols - is easily applied systematically to hand-crafted parsers. It suffers from a number of disadvantages, however:

- It is quite expensive, since each call to a parsing routine is effectively preceded by two time-consuming operations - the dynamic construction of a set object, and the parameter passing operation itself - operations which turn out not to have been required if the source being translated is correct.

- If, as often happens, seemingly superfluous symbols like semicolons are omitted from the source text, the resynchronization process can be overly severe.

Thus the scheme is usually adapted somewhat, often in the light of experience gained by observing typical user errors. A study of the source code for such parsers on the source diskette will reveal examples of the following useful variations on the basic scheme:

- In those many places where "weak" separators are found in constructs involving iterations, such as

```
VarDeclarations     =  "VAR" OneVar { "," OneVar } ";"
```

```
CompoundStatement   =  "BEGIN" Statement { ";" Statement } "END" .
Term                =  Factor  { MulOp  Factor } .
```

the iteration is started as long as the parser detects the presence of the weak separator *or* a valid symbol that would follow it in that context (of course, appropriate errors are reported if the separator has been omitted). This has the effect of "inserting" such missing separators into the stream of symbols being parsed, and proves to be a highly effective enhancement to the basic technique.

- Places where likely errors are expected - such as confusion between the ":=" and "=" operators, or attempting to provide an integer expression rather than a Boolean comparison expression in an *IfStatement* or *WhileStatement* - are handled in an *ad-hoc* way.

- Many sub-parsers do not need to make use of the prologue and epilogue calls to the `test` routine. In particular, there is no need to do this in routines like those for *IfStatement*, *WhileStatement* and so on, which have been introduced mainly to enhance the modularity of *Statement*.

- The Modula-2 and Pascal implementations nest their parsing routines as tightly as possible. Not only does this match the embedded nature of the grammar very nicely, it also reduces the number of parameters that have to be passed around.

Because of the inherent cost in the follower-set based approach to error recovery, some compiler writers make use of simpler schemes that can achieve very nearly the same degree of success at far less cost. One of these, suggested by Wirth (1986, 1996), is based on the observation that the symbols passed as members of follower sets to high level parsing routines - such as *Block* - effectively become members of every follower set parameter computed thereafter. When one finally gets to parse a *Statement*, for example, the set of stopping symbols used to establish synchronization at the start of the *Statement* routine is the union of FIRST(*Statement*) + FOLLOW(*Program*) + FOLLOW(*Block*), while the set of stopping symbols used to establish synchronization at the end of *Statement* is the union of FOLLOW(*Statement*) + FOLLOW(*Program*) + FOLLOW(*Block*). Furthermore, if we treat the semicolon that separates statements as a "weak" separator, as previously discussed, no great harm is done if the set used to establish synchronization at the end of *Statement* also includes the elements of FIRST(*Statement*).

Careful consideration of the `SCAN_symtypes` enumeration introduced in section 14.4.1 will reveal that the values have been ordered so that the following patterns hold to a high degree of accuracy:

```
SCAN_unknown .. SCAN_lbracket,     Miscellaneous
SCAN_times, SCAN_slash,            FOLLOW(Factor)
SCAN_plus, SCAN_minus,             FOLLOW(Term)
SCAN_eqlsym .. SCAN_geqsym,        FOLLOW(Expression1) in Condition
SCAN_thensym, SCAN_dosym,          FOLLOW(Condition)
SCAN_rbracket .. SCAN_comma,       FOLLOW(Expression)
SCAN_lparen, .. SCAN_identifier,   FIRST(Factor)
SCAN_coendsym, SCAN_endsym,        FOLLOW(Statement)
SCAN_ifsym .. SCAN_signalsym,      FIRST(Statement)
SCAN_semicolon,                    FOLLOW(Block)
SCAN_beginsym .. SCAN_funcsym,     FIRST(Block)
SCAN_period,                       FOLLOW(Program)
SCAN_progsym,                      FIRST(Program)
SCAN_eofsym
```

The argument now goes that, with this carefully ordered enumeration, virtually all of the tests of the form

```
Sym ∈ SynchronizationSet
```

can be accurately replaced by tests of the form

$$Sym \geq SmallestElement(SynchronizationSet)$$

and that synchronization at crucial points in the grammar can be achieved by using a routine developed on the lines of

```
void synchronize(SCAN_symtypes SmallestElement, int errorcode)
{ if (SYM.sym >= SmallestElement) return;
  reporterror(errorcode);
  do { getsym(); } while (SYM.sym < SmallestElement);
}
```

The way in which this idea could be used is exemplified in a routine for parsing Clang statements.

```
void Statement(void)
// Statement = [ CompoundStatement | Assignment | IfStatement
//             | WhileStatement | WriteStatement | ReadStatement ] .
{ synchronize(SCAN_identifier, 15);
  // We shall return correctly if SYM.sym is a semicolon or END (empty statement)
  // or if we have synchronized (prematurely) on a symbol that really follows
  // a Block
  switch (SYM.sym)
  { case SCAN_identifier: Assignment(); break;
    case SCAN_ifsym:      IfStatement(); break;
    case SCAN_whilesym:   WhileStatement(); break;
    case SCAN_writesym:   WriteStatement(); break;
    case SCAN_readsym:    ReadStatement(); break;
    case SCAN_beginsym:   CompoundStatement(); break;
    default:              return;
  }
  synchronize(SCAN_endsym, 32);
  // In some situations we shall have synchronized on a symbol that can start
  // a further Statement, but this should be handled correctly from the call
  // made to Statement from within CompoundStatement
}
```

It turns out to be necessary to replace some other set inclusion tests, by providing predicate functions exemplified by

```
bool inFirstStatement(SCAN_symtypes Sym)
// Returns true if Sym can start a Statement
{ return (Sym == SCAN_identifier || Sym == SCAN_beginsym ||
          Sym >= SCAN_ifsym && Sym <= SCAN_signalsym);
}
```

Complete parsers using this ingenious scheme are to be found on the source diskette. However, the idea is fairly fragile. Symbols do not always fall uniquely into only one of the FIRST or FOLLOW sets, and in large languages there may be several keywords (like `END`, `CASE` and `OF` in Pascal) that can appear in widely different contexts. If new keywords are added to an evolving language, great care has to be taken to maintain the optimum ordering; if a token value is misplaced, error recovery would be badly affected.

The scheme can be made more robust by declaring various synchronization set constants, without requiring their elements to have contiguous values. This is essentially the technique used in Coco/R generated recovery schemes, and adapting it to hand-crafted parsers is left as an interesting exercise for the reader.

14.6.2 Syntax error handling in Coco/R generated parsers

The way in which a Cocol description of a grammar is augmented to indicate where synchronization should be attempted has already been discussed in section 12.4.2. To be able to achieve optimal use of the basic facilities offered by the use of the `SYNC` and `WEAK` directives calls for some ingenuity. If too many `SYNC` directives are introduced, the error recovery achievable with

the use of WEAK can actually deteriorate, since the union of all the SYNC symbol sets tends to become the entire universe. Below we show a modification of the grammar in section 8.7.2 that has been found to work quite well, and draw attention to the use of two places (in the productions for *Condition* and *Term*) where an explicit call to the error reporting interface has been used to handle situations where one wishes to be lenient in the treatment of missing symbols.

```
PRODUCTIONS /* some omitted to save space */
  Clang             = "PROGRAM" identifier WEAK ";" Block "." .
  Block             = SYNC { ( ConstDeclarations | VarDeclarations ) SYNC }
                      CompoundStatement .
  OneConst          = identifier WEAK "=" number ";" .
  VarDeclarations   = "VAR" OneVar { WEAK "," OneVar } ";" .
  CompoundStatement = "BEGIN" Statement { WEAK ";" Statement } "END" .
  Statement         = SYNC [   CompoundStatement | Assignment
                             | IfStatement       | WhileStatement
                             | ReadStatement     | WriteStatement ] .
  Assignment        = Variable ":=" Expression SYNC .
  Condition         = Expression ( RelOp Expression | (. SynError(91); .) ) .
  ReadStatement     = "READ" "(" Variable { WEAK "," Variable } ")" .
  WriteStatement    = "WRITE"
                      [ "(" WriteElement { WEAK "," WriteElement }  ")" ] .
  Term              = Factor { ( MulOp | (. SynError(92); .) ) Factor } .
```

14.6.3 Constraint analysis and static semantic error handling

We have already had cause to remark that the boundary between syntactic and semantic errors can be rather vague, and that there are features of real computer languages that cannot be readily described by context-free grammars. To retain the advantages of simple one-pass compilation when we include semantic analysis, and start to attach meaning to our identifiers, usually requires that the "declaration" parts of a program come before the "statement" parts. This is easily enforced by a context-free grammar, all very familiar to a Modula-2, Pascal or C programmer, and seems quite natural after a while. But it is only part of the story. Even if we insist that declarations precede statements, a context-free grammar is still unable to specify that only those identifiers which have appeared in the declarations (so-called *defining occurrences*) may appear in the statements (in so-called *applied occurrences*). Nor is a context-free grammar powerful enough to specify such constraints as insisting that only a variable identifier can be used to denote the target of an assignment statement, or that a complete array cannot be assigned to a scalar variable. We might be tempted to write productions that seem to capture these constraints:

```
  Clang             =  "PROGRAM" ProgIdentifier ";" Block "." .
  Block             =  { ConstDeclarations | VarDeclarations }
                       CompoundStatement .
  ConstDeclarations =  "CONST" OneConst { OneConst } .
  OneConst          =  ConstIdentifier "=" number ";" .
  VarDeclarations   =  "VAR" OneVar { "," OneVar } ";" .
  OneVar            =  ScalarVarIdentifier | ArrayVarIdentifier UpperBound .
  UpperBound        =  "[" number "]" .
  Assignment        =  Variable ":=" Expression .
  Variable          =  ScalarVarIdentifier | ArrayVarIdentifier "[" Expression "]" .
  ReadStatement     =  "READ" "(" Variable { "," Variable } ")" .
  Expression        =  ( "+" Term | "-" Term | Term ) { AddOp Term } .
  Term              =  Factor { MulOp Factor } .
  Factor            =    ConstIdentifier | Variable | number
                       | "(" Expression ")" .
```

This would not really get us very far, since all identifiers are lexically equivalent! We could attempt to use a context-sensitive grammar to overcome such problems, but that turns out to be unnecessarily complicated, for they are easily solved by leaving the grammar as it was, adding attributes in the form of context conditions, and using a symbol table.

Demanding that identifiers be declared in such a way that their static semantic attributes can be recorded in a symbol table, whence they can be retrieved at any future stage of the analysis, is not nearly as tedious as users might at first imagine. It is clearly a semantic activity, made easier by a syntactic association with keywords like CONST, VAR and PROGRAM.

Setting up a symbol table may be done in many ways. If one is interested merely in performing the sort of constraint analysis suggested earlier for a language as simple as Clang we may begin by noting that identifiers designate objects that are restricted to one of three simple varieties - namely *constant*, *variable* and *program*. The only apparent complication is that, unlike the other two, a variable identifier can denote either a simple scalar, or a simple linear array. A simple table handler can then be developed with a class having a public interface like the following:

```
const int TABLE_alfalength = 15; // maximum length of identifiers
typedef char TABLE_alfa[TABLE_alfalength + 1];

enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs };

struct TABLE_entries {
  TABLE_alfa name;             // identifier
  TABLE_idclasses idclass;     // class
  bool scalar;                 // distinguish arrays from scalars
};

class TABLE {
  public:
    TABLE(REPORT *R);
    // Initializes symbol table

    void enter(TABLE_entries &entry);
    // Adds entry to symbol table

    void search(char *name, TABLE_entries &entry, bool &found);
    // Searches table for presence of name.  If found then returns entry

    void printtable(FILE *lst);
    // Prints symbol table for diagnostic purposes
};
```

An augmented parser must construct appropriate `entry` structures and enter these into the table when identifiers are first recognized by the routine that handles the productions for *OneConst* and *OneVar*. Before identifiers are accepted by the routines that handle the production for *Designator* they are checked against this table for non-declaration of `name`, abuse of `idclass` (such as trying to assign to a constant) and abuse of `scalar` (such as trying to subscript a constant, or a scalar variable). The interface suggested here is still inadequate for the purposes of code generation, so we delay further discussion of the symbol table itself until the next section.

However, we may take the opportunity of pointing out that the way in which the symbol table facilities are used depends rather critically on whether the parser is being crafted by hand, or by using a parser generator. We draw the reader's attention to the production for *Factor*, which we have written

```
Factor = Designator | number | "(" Expression ")" .
```

rather than the more descriptive

```
Factor = ConstIdentifier | Variable | number | "(" Expression ")" .
```

which does not satisfy the LL(1) constraints. In a hand-crafted parser we are free to use semantic information to drive the parsing process, and to break this LL(1) conflict, as the following extract from such a parser will show.

```
void Factor(symset followers)
// Factor = Variable | ConstIdentifier | Number | "(" Expression ")" .
// Variable = Designator .
{ TABLE_entries entry;
  bool found;
  test(FirstFactor, followers, 14);              // Synchronize
  switch (SYM.sym)
  { case SCAN_identifier:
      Table->search(SYM.name, entry, found);     // Look it up
      if (!found) Report->error(202);            // Undeclared identifier
```

```
      if (entry.idclass = TABLE_consts) GetSym(); // ConstIdentifier
      else Designator(entry, followers, 206);     // Variable
      break;
    case SCAN_number:
      GetSym(); break;
    case SCAN_lparen:
      GetSym(); Expression(symset(SCAN_rparen) + followers);
      accept(SCAN_rparen, 17); break;
    default:                                      // Synchronized on a
      Report->error(14); break;                   // follower instead
  }
}
```

In a Coco/R generated parser some other way must be found to handle the conflict. The generated parser will always set up a call to parse a *Designator*, and so the distinction must be drawn at that stage. The following extracts from an attributed grammar shows one possible way of doing this.

```
Factor
=                          (. int value; TABLE_entries entry; .)
     Designator<classset(TABLE_consts, TABLE_vars), entry>
   | Number<value>
   | "(" Expression ")" .
```

Notice that the *Designator* routine is passed the set of `idclasses` that are acceptable in this context. The production for *Designator* needs to check quite a number of context conditions:

```
Designator<classset allowed, TABLE_entries &entry>
=                          (. TABLE_alfa name;
                              bool isvariable, found; .)
   Ident<name>             (. Table->search(name, entry, found);
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              isvariable = entry.idclass == TABLE_vars; .)
   ( "["                   (. if (!isvariable || entry.scalar) SemError(204); .)
     Expression "]"
     |                     (. if (isvariable && !entry.scalar) SemError(205); .)
   ) .
```

Other variations on this theme are possible. One of these is interesting in that it effectively uses semantic information to drive the parser, returning prematurely if it appears that a subscript should not be allowed:

```
Designator<classset allowed, TABLE_entries &entry>
=                          (. TABLE_alfa name;
                              bool found; .)
   Ident<name>             (. Table->search(name, entry, found);
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              if (entry.idclass != TABLE_vars) return; .)
   ( "["                   (. if (entry.scalar) SemError(204); .)
     Expression "]"
     |                     (. if (!entry.scalar) SemError(205); .)
   ) .
```

As an example of how these ideas combine in the reporting of incorrect programs we present a source listing produced by the hand-crafted parser found on the source diskette:

```
 1 : PROGRAM Debug
 2 :   CONST
 2 :        ^; expected
 3 :     TooBigANumber = 328000;
 3 :                     ^Constant out of range
 4 :     Zero := 0;
 4 :           ^:= in wrong context
 5 :   VAR
 6 :     Valu, Smallest, Largest, Total;
 7 :   CONST
 8 :     Min = Zero;
 8 :              ^Number expected
 9 :   BEGIN
10 :     Total := Zero;
11 :     IF Valu THEN;
11 :                ^Relational operator expected
12 :     READ (Valu); IF Valu > Min DO WRITE(Valu);
```

```
12 :                                  ^THEN expected
13 :     Largest := Valu; Smallest = Valu;
13 :                                ^:= expected
14 :     WHILE Valu <> Zero DO
15 :       BEGIN
16 :         Total := Total + Valu
17 :         IF Valu > = Largest THEN Largest := Value;
17 :           ^; expected
17 :                    ^Invalid factor
17 :                                                  ^Undeclared identifier
18 :         IF Valu < Smallest THEN Smallest := Valu;
19 :         READLN(Valu); IF Valu > Zero THEN WRITE(Valu)
19 :                ^Undeclared identifier
20 :       END;
21 :     WRITE('TOTAL:', Total, ' LARGEST:', Largest);
22 :     WRITE('SMALLEST: , Smallest)
22 :                                  ^Incomplete string
23 :   END.
23 :      ^) expected
```

Exercises

14.37 Submit the incorrect program given above to a Coco/R generated parser, and compare the quality of error reporting and recovery with that achieved by the hand-crafted parser.

14.38 At present the error messages for the hand-crafted system are reported one symbol *after* the point where the error was detected. Can you find a way of improving on this?

14.39 A disadvantage of the error recovery scheme used here is that a user may not realize which symbols have been skipped. Can you find a way to mark some or all of the symbols skipped by `test`? Has `test` been used in the best possible way to facilitate error recovery?

14.40 If, as we have done, all error messages after the first at a given point are suppressed, one might occasionally find that the quality of error message deteriorates - "early" messages might be less apposite than "later" messages might have been. Can you implement a better method than the one we have? (Notice that the *Followers* parameter passed to a sub-parser for *S* includes not only the genuine FOLLOW(*S*) symbols, but also further *Beacons*.)

14.41 If you study the code for the hand-crafted parser carefully you will realize that *Identifier* effectively appears in all the *Follower* sets? Is this a good idea? If not, what alterations are needed?

14.42 Although not strictly illegal, the appearance of a semicolon in a program immediately following a `DO` or `THEN`, or immediately preceding an `END` may be symptomatic of omitted code. Is it possible to warn the user when this has occurred, and if so, how?

14.43 The error reporter makes no real distinction between context-free and semantic or context-sensitive errors. Do you suppose it would be an improvement to try to do this, and if so, how could it be done?

14.44 Why does this parser not allow you to assign one array completely to another array? What modifications would you have to make to the context-free grammar to permit this? How would the constraint analysis have to be altered?

14.45 In Topsy - at least as it is used in the example program of Exercise 8.25 - all "declarations" seem to precede "statements". In C++ it is possible to declare variables at the point where they are first needed. How would you define Topsy to support the mingling of declarations and statements?

14.46 One school of thought maintains that in a statement like a Modula-2 `FOR` loop, the control variable should be implicitly declared at the start of the loop, so that it is truly local to the loop. It should also not be possible to alter the value of the control variable within the loop. Can you extend your parser and symbol table handler to support these ideas?

14.47 Exercises 14.21 through 14.36 suggested many syntactic extensions to Clang or Topsy. Extend your parsers so that they incorporate error recovery and constraint analysis for all these extensions.

14.48 Experiment with error recovery mechanisms that depend on the ordering of the `SCAN_symtypes` enumeration, as discussed in section 14.6.1. Can you find an ordering that works adequately for Topsy?

14.7 The symbol table handler

In an earlier section we claimed that it would be advantageous to split our compiler into distinct phases for syntax/constraint analysis and code generation. One good reason for doing this is to isolate the machine dependent part of compilation as far as possible from the language analysis. The degree to which we have succeeded may be measured by the fact that we have not yet made any mention of what sort of object code we are trying to generate.

Of course, any interface between source and object code must take cognizance of data-related concepts like *storage, addresses* and *data representation*, as well as control-related ones like *location counter, sequential execution* and *branch instruction*, which are fundamental to nearly all machines on which programs in our imperative high-level languages execute. Typically, machines allow some operations which simulate arithmetic or logical operations on data bit patterns which simulate numbers or characters, these patterns being stored in an array-like structure of *memory*, whose elements are distinguished by *addresses*. In high-level languages these addresses are usually given mnemonic names. The context-free syntax of many high-level languages, as it happens, rarely seems to draw a distinction between the "address" for a variable and the "value" associated with that variable, and stored at its address. Hence we find statements like

```
X := X + 4
```

in which the `X` on the left of the `:=` operator actually represents an address, (sometimes called the *L-value* of `X`) while the `X` on the right (sometimes called the *R-value* of `X`) actually represents the value of the quantity currently residing at the same address. Small wonder that mathematically trained beginners sometimes find the assignment notation strange! After a while it usually becomes second nature - by which time notations in which the distinction is made clearer possibly only confuse still further, as witness the problems beginners often have with pointer types in C++ or Modula-2, where `*P` or `P^` (respectively) denote the explicit value residing at the explicit address `P`. If we relate this back to the productions used in our grammar, we would find that each `X` in the above assignment was syntactically a *Designator*. Semantically these two designators are very different - we shall refer to the one that represents an address as a *Variable Designator*, and to the one that represents a value as a *Value Designator*.

To perform its task, the code generation interface will require the extraction of further information associated with user-defined identifiers and best kept in the symbol table. In the case of constants we need to record the associated values, and in the case of variables we need to record the associated addresses and storage demands (the elements of array variables will occupy a contiguous

block of memory). If we can assume that our machine incorporates a "linear array" model of memory, this information is easily added as the variables are declared.

Handling the different sorts of entries that need to be stored in a symbol table can be done in various ways. In a object-oriented class-based implementation one might define an abstract base class to represent a generic type of entry, and then derive classes from this to represent entries for variables or constants (and, in due course, records, procedures, classes and any other forms of entry that seem to be required). The traditional way, still required if one is hosting a compiler in a language that does not support inheritance as a concept, is to make use of a variant record (in Modula-2 terminology) or union (in C++ terminology). Since the class-based implementation gives so much scope for exercises, we have chosen to illustrate the variant record approach, which is very efficient, and quite adequate for such a simple language. We extend the declaration of the `TABLE_entries` type to be

```
struct TABLE_entries {
  TABLE_alfa name;          // identifier
  TABLE_idclasses idclass;  // class
  union {
    struct {
      int value;
    } c;                    // constants
    struct {
      int size, offset;     // number of words, relative address
      bool scalar;          // distinguish arrays
    } v;                    // variables
  };
};
```

The way in which the symbol table is constructed can be illustrated with reference to the relevant parts of a Cocol specification for handling *OneConst* and *OneVar*:

```
OneConst
=                             (. TABLE_entries entry; .)
   Ident<entry.name>          (. entry.idclass = TABLE_consts; .)
   WEAK "="
   Number<entry.c.value> ";"  (. Table->enter(entry); .) .

OneVar<int &framesize>
=                             (. TABLE_entries entry;
                                 entry.idclass = TABLE_vars;
                                 entry.v.size = 1; entry.v.scalar = true;
                                 entry.v.offset = framesize + 1; .)
   Ident<entry.name>
   [ UpperBound<entry.v.size> (. entry.v.scalar = false; .)
   ]                          (. Table->enter(entry);
                                 framesize += entry.v.size; .) .

UpperBound<int &size>
=  "[" Number<size> "]"       (. size++; .) .

Ident<char *name>
=  identifier                 (. LexName(name, TABLE_alfalength); .) .
```

Here `framesize` is a simple count, which is initialized to zero at the start of parsing a *Block*. It keeps track of the number of variables declared, and also serves to define the addresses which these variables will have relative to some known location in memory when the program runs. A trivial modification gets around the problem if it is impossible or inconvenient to use zero-based addresses in the real machine.

Programming a symbol table handler for a language as simple as ours can be correspondingly simple. On the source diskette can be found such implementations, based on the idea that the symbol table can be stored within a fixed length array. A few comments on implementation techniques will guide the reader who wishes to study this code:

- The table is set up so that the entry indexed by zero can be used as a sentinel in a simple

sequential search by `search`. Although this is inefficient, it is adequate for prototyping the system.

- A call to `Table->search(name, entry, found)` will always return with a well defined value for `entry`, even if the `name` had never been declared. Such undeclared identifiers will seem to have an effective `idclass = TABLE_progs`, which will be semantically unacceptable everywhere, thus ensuring that incorrect code can never be generated.

Exercises

14.49 How would you check that no identifier is declared more than once?

14.50 Identifiers that are undeclared by virtue of mistyped declarations tend to be annoying, for they result in many subsequent errors being reported. Perhaps in languages as simple as ours one could assume that all undeclared identifiers should be treated as variables, and entered as such in the symbol table at the point of first reference. Is this a good idea? Can it easily be implemented? What happens if arrays are undeclared?

14.51 Careful readers may have noticed that a Clang array declaration is different from a C++ one - the bracketed number in Clang specifies the highest permitted index value, rather than the array length. This has been done so that one can declare variables like

```
VAR Scalar, List[10], VeryShortList[0];
```

How would you modify Clang and Topsy to use C++ semantics, where the declaration of `VeryShortList` would have to be forbidden?

14.52 The names of identifiers are held within the symbol table as fixed length strings, truncated if necessary. It may seem unreasonable to expect compilers (especially written in Modula-2 or Pascal, which do not have dynamic strings as standard types) to cater for identifiers of any length, but too small a limitation on length is bound to prove irksome sooner or later, and too generous a limitation simply wastes valuable space when, as so often happens, users choose very short names. Develop a variation on the symbol table handler that allocates the name fields dynamically, to be of the correct size. (This can, of course, also be done in Modula-2.) Making table entries should be quite simple; searching for them may call for a little more ingenuity.

14.53 A simple sequential search algorithm is probably perfectly adequate for the small Clang programs that one is likely to write. It becomes highly inefficient for large applications. It is far more efficient to store the table in the form of a binary search tree, of the sort that you may have encountered in other courses in Computer Science. Develop such an implementation, noting that it should not be necessary to alter the public interface to the table class.

14.54 Yet another approach is to construct the symbol table using a hash table, which probably yields the shortest times for retrievals. Hash tables were briefly discussed in Chapter 7, and should also be familiar from other courses you may have taken in Computer Science. Develop a hash table implementation for your Clang or Topsy compiler.

14.55 We might consider letting the scanner interact with the symbol table. Consider the implications of developing a scanner that stores the strings for identifiers and string literals in a string table, as suggested in Exercise 6.6 for the assemblers of Chapter 6.

14.56 Develop a symbol table handler that utilizes a simple class hierarchy for the possible types of entries, inheriting appropriately from a suitable base class. Once again, construction of such a table should prove to be straightforward, regardless of whether you use a linear array, tree, or hash table as the underlying storage structure. Retrieval might call for more ingenuity, since C++ does not provide syntactic support for determining the exact class of an object that has been statically declared to be of a base class type.

14.8 Other aspects of symbol table management - further types

It will probably not have escaped the reader's attention, especially if he or she has attempted the exercises in the last few sections, that compilers for languages which handle a wide variety of types, both "standard" and "user defined", must surely take a far more sophisticated approach to constructing a symbol table and to keeping track of storage requirements than anything we have seen so far. Although the nature of this text does not warrant a full discussion of this point, a few comments may be of interest, and in order.

In the first place, a compiler for a block-structured language will probably organize its symbol table as a collection of dynamically allocated trees, with one root for each level of nesting. Although using simple binary trees runs the risk of producing badly unbalanced trees, this is unlikely. Except for source programs which are produced by program generators, user programs tend to introduce identifiers with fairly random names; few compilers are likely to need really sophisticated tree constructing algorithms.

Secondly, the nodes in the trees will be fairly complex record structures. Besides the obvious links to other nodes in the tree, there will probably be pointers to other dynamically constructed nodes, which contain descriptions of the types of the identifiers held in the main tree.

Thus a Pascal declaration like

```
VAR
  Matrix : ARRAY [1 .. 10, 2 .. 20] OF SET OF CHAR;
```

might result in a structure that can be depicted something like that of Figure 14.2.

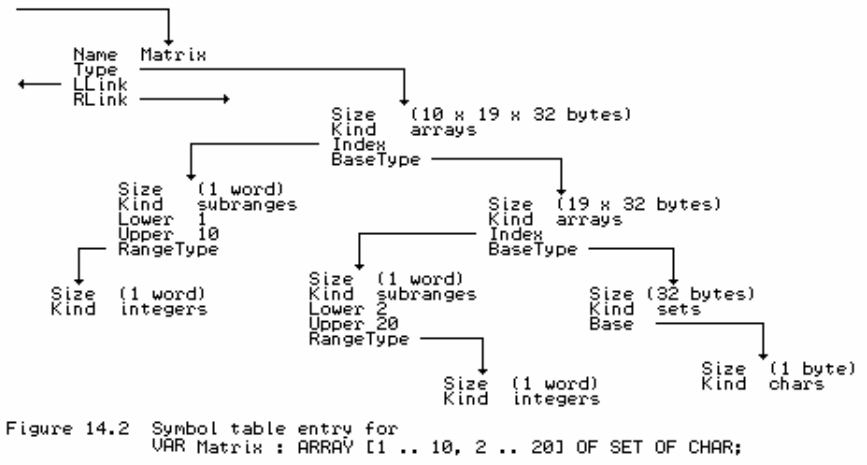


Figure 14.2 Symbol table entry for
VAR Matrix : ARRAY [1 .. 10, 2 .. 20] OF SET OF CHAR;

We may take this opportunity to comment on a rather poorly defined area in Pascal, one that came in for much criticism. Suppose we were to declare

```
TYPE
  LISTS = ARRAY [1 .. 10] OF CHAR;
VAR
  X : LISTS;
  A : ARRAY [1 .. 10] OF CHAR;
  B : ARRAY [1 .. 10] OF CHAR;
  Z : LISTS;
```

`A` and `B` are said to be of **anonymous type**, but to most people it would seem obvious that `A` and `B` are of the same type, implying that an assignment of the form `A := B` should be quite legal, and, furthermore, that `X` and `Z` would be of the same type as `A` and `B`. However, some compilers will be satisfied with mere **structural equivalence** of types before such an assignment would be permitted, while others will insist on so-called **name equivalence**. The original Pascal Report did not specify which was standard.

In this example `A, B, X` and `Z` all have structural equivalence. `X` and `Z` have name equivalence as well, as they have been specified in terms of a named type `LISTS`.

With the insight we now have we can see what this difference means from the compiler writer's viewpoint. Suppose `A` and `B` have entries in the symbol table pointed to by `ToA` and `ToB` respectively. Then for name equivalence we should insist on `ToA^.Type` and `ToB^.Type` being the same (that is, their `Type` pointers address the same descriptor), while for structural equivalence we should insist on `ToA^.Type^` and `ToA^.Type^` being the same (that is, their `Type` pointers address descriptors that have the same structure).

Further reading and exploration

We have just touched on the tip of a large and difficult iceberg. If one adds the concept of types and type constructors into a language, and insists on strict type-checking, the compilers become much larger and harder to follow than we have seen up till now. The energetic reader might like to follow up several of the ideas which should now come to mind. Try a selection of the following, which are deliberately rather vaguely phrased.

14.57 Just how do real compilers deal with symbol tables?

14.58 Just how do real compilers keep track of type checking? Why should name equivalence be easier to handle than structural equivalence?

14.59 Why do some languages simply forbid the use of "anonymous types", and why don't more languages forbid them?

14.60 How do you suppose compilers keep track of storage allocation for `struct` or `RECORD` types, and for `union` or variant record types?

14.61 Find out how storage is managed for dynamically allocated variables in language like C++, Pascal, or Modula-2.

14.62 How does one cope with arrays of variable (dynamic) length in subprograms?

14.63 Why can we easily allow the declaration of a pointer type to precede the definition of the type it points to, even in a one-pass system? For example, in Modula-2 we may write

```
TYPE
  LINKS = POINTER TO NODES (* NODES not yet seen *);
  NODES = RECORD
            ETC  : JUNK;
            Link : LINKS;
            . . .
```

14.64 Brinch Hansen did not like the Pascal subrange type because it seems to lead to ambiguities (for example, a value of `34` can be of type `0 .. 45`, and also of type `30 .. 90` and so on), and so omitted them from Edison. Interestingly, a later Wirth language, Oberon, omits them as well. How might Pascal and Modula-2 otherwise have introduced the subrange concept, how could we overcome Brinch Hansen's objections, and what is the essential point that he seems to have overlooked in discarding them?

14.65 One might accuse the designers of Pascal, Modula-2 and C of making a serious error of judgement - they do not introduce a string type as standard, but rely on programmers to manipulate arrays of characters, and to use error prone ways of recognizing the end, or the length, of a string. Do you agree? Discuss whether what they offer in return is adequate, and if not, why not. Suggest why they might deliberately not have introduced a string type.

14.66 Brinch Hansen did not like the Pascal variant record (or `union`). What do such types allow one to do in Pascal which is otherwise impossible, and why should it be necessary to provide the facility to do this? How else are these facilities catered for in Modula-2, C and C++? Which is the better way, and why? Do the ideas of type extension, as found in Oberon, C++ and other "object oriented" languages provide even better alternatives?

14.67 Many authors dislike pointer types because they allow "insecure" programming". What is meant by this? How could the security be improved? If you do not like pointer types, can you think of any alternative feature that would be more secure?

There is quite a lot of material available on these subjects in many of the references cited previously. Rather than give explicit references, we leave the Joys of Discovery to the reader.

15 A SIMPLE COMPILER - THE BACK END

After the front end has analysed the source code, the back end of a compiler is responsible for synthesizing object code. The critical reader will have realized that code generation, of any form, implies that we consider the semantics of our language and of our target machine, and the interaction between them, in far more detail than we have done until now. Indeed, we have made no real attempt to define what programs written in Clang or Topsy "mean", although we have tacitly assumed that the reader has quite an extensive knowledge of imperative languages, and that we could safely draw on this.

15.1 The code generation interface

In considering the interface between analysis and code generation it will again pay to aim for some degree of machine independence. Generation of code should take place without too much, if any, knowledge of how the analyser works. A common technique for achieving this seemingly impossible task is to define a hypothetical machine, with instruction set and architecture convenient for the execution of programs of the source language, but without being too far removed from the actual system for which the compiler is required. The action of the interface routines will be to translate the source program into an equivalent sequence of operations for the hypothetical machine. Calls to these routines can be embedded in the parser without overmuch concern for how the final generator will turn the operations into object code for the target machine. Indeed, as we have already mentioned, some interpretive systems hand such operations over directly to an interpreter without ever producing real machine code.

The concepts of the meaning of an expression, and of assignment of the "values" of expressions to locations in memory labelled with the "addresses" of variables are probably well understood by the reader. As it happens, such operations are very easily handled by assuming that the hypothetical machine is stack-based, and translating the normal infix notation used in describing expressions into a postfix or Polish equivalent. This is then easily handled with the aid of an evaluation stack, the elements of which are either addresses of storage locations, or the values found at such addresses. These ideas will probably be familiar to readers already acquainted with stack- based machines like the Hewlett-Packard calculator. Furthermore, we have already examined a model of such a machine in section 2.4, and discussed how expressions might be converted into postfix notation in section 11.1.

A little reflection on this theme will suggest that the public interface of such a code generation class might take the form below.

```
enum CGEN_operators {
  CGEN_opadd, CGEN_opsub, CGEN_opmul, CGEN_opdvd, CGEN_opeql,
  CGEN_opneq, CGEN_oplss, CGEN_opgeq, CGEN_opgtr, CGEN_opleq
};

typedef short CGEN_labels;

class CGEN {
  public:
    CGEN_labels undefined;    // for forward references

    CGEN(REPORT *R);
    // Initializes code generator
```

```
    void negateinteger(void);
    // Generates code to negate integer value on top of evaluation stack

    void binaryintegerop(CGEN_operators op);
    // Generates code to pop two values A,B from stack and push value A op B

    void comparison(CGEN_operators op);
    // Generates code to pop two values A,B from stack; push Boolean value A op B

    void readvalue(void);
    // Generates code to read an integer; store on address found on top-of-stack

    void writevalue(void);
    // Generates code to pop and then output the value at top-of-stack

    void newline(void);
    // Generates code to output line mark

    void writestring(CGEN_labels location);
    // Generates code to output string stored from known location

    void stackstring(char *str, CGEN_labels &location);
    // Stores str in literal pool in memory and returns its location

    void stackconstant(int number);
    // Generates code to push number onto evaluation stack

    void stackaddress(int offset);
    // Generates code to push address for known offset onto evaluation stack

    void subscript(void);
    // Generates code to index an array and check that bounds are not exceeded

    void dereference(void);
    // Generates code to replace top-of-stack by the value stored at the
    // address currently stored at top-of-stack

    void assign(void);
    // Generates code to store value currently on top-of-stack on the
    // address stored at next-to-top, popping these two elements

    void openstackframe(int size);
    // Generates code to reserve space for size variables

    void leaveprogram(void);
    // Generates code needed as a program terminates (halt)

    void storelabel(CGEN_labels &location);
    // Stores address of next instruction in location for use in backpatching

    void jump(CGEN_labels &here, CGEN_labels destination);
    // Generates unconditional branch from here to destination

    void jumponfalse(CGEN_labels &here, CGEN_labels destination);
    // Generates branch from here to destination, conditional on the Boolean
    // value currently on top of the evaluation stack, popping this value

    void backpatch(CGEN_labels location);
    // Stores the current location counter as the address field of the branch
    // instruction assumed to be held in an incomplete form at location

    void dump(void);
    // Generates code to dump the current state of the evaluation stack

    void getsize(int &codelength, int &initsp);
    // Returns length of generated code and initial stack pointer

    int gettop(void);
    // Returns the current location counter
};
```

As usual, there are several points that need further comment and explanation:

- The code generation routines have been given names that might suggest that they actually *perform* operations like `jump`. They only *generate* code for such operations, of course.

- There is an unavoidable interaction between this class and the machine for which code is to be generated - the implementation will need to import the machine address type, and we have

seen fit to export a routine (`getsize`) that will allow the compiler to determine the amount of code generated.

- Code for data manipulation on such a machine can be generated by making calls on routines like `stackconstant, stackaddress, stackstring, subscript` and `dereference` for storage access; by calls to routines like `negateinteger` and `binaryintegerop` to generate code to perform simple arithmetic; and finally by calls to `assign` to handle the familiar assignment process.

  For example, compilation of the Clang assignment statement

```
A := 4 + List[5]
```

  (where `List` has 14 elements) should result in the following sequence of code generator routine calls

```
stackaddress(offset of A)
  stackconstant(4)
  stackaddress(offset of List[0])
  stackconstant(5)
  stackconstant(14)
  subscript
  dereference
  binaryintegerop(CGEN_opadd)
assign
```

- The address associated with an array in the symbol table will denote the offset of the first element of the array (the zero-subscript one) from some known "base" at run-time. Our arrays are very simple indeed. They have only one dimension, a size `N` fixed at compile-time, a fixed lower subscript bound of zero, and can easily be handled after allocating them `N` consecutive elements in memory. Addressing an individual element at run time is achieved by computing the value of the subscripting expression, and adding this to (or, on a stack implementation, subtracting it from) the address of the first element in the array. In the interests of safety we shall insist that all subscripting operations incorporate range checks (this is, of course, not done in C++).

- To generate code to handle simple I/O operations we can call on the routines `readvalue, writevalue, writestring` and `newline`.

- To generate code to allow comparisons to be effected we call on `comparison`, suitable parameterized according to the test to be performed.

- Control statements are a little more interesting. In the type of machine being considered it is assumed that machine code will be executed in the order in which it was generated, except where explicit "branch" operations occur. Although our simple language does not incorporate the somewhat despised `GOTO` statement, this maps very closely onto real machine code, and must form the basis of code generated by higher level control statements. The transformation is, of course, easily automated, save for the familiar problem of forward references. In our case there are two source statements that give rise to these. Source code like

```
IF Condition THEN Statement
```

  should lead to object code of the more fundamental form

```
        code for Condition
        IF NOT Condition THEN GOTO LAB END
        code for Statement
LAB     continue
```

and the problem is that when we get to the stage of generating `GOTO LAB` we do not know the address that will apply to `LAB`. Similarly, the source code

```
WHILE Condition DO Statement
```

should lead to object code of the form

```
LAB    code for Condition
       IF NOT Condition THEN GOTO EXIT END
       code for Statement
       GOTO LAB
EXIT   continue
```

Here we should know the address of `LAB` as we start to generate the code for *Condition*, but we shall not know the address of `EXIT` when we get to the stage of generating `GOTO EXIT`.

In general the solution to this problem might require the use of a two-pass system. However, we shall assume that we are developing a one-pass load-and-go compiler, and that the generated code is all in memory, or at worst on a random access file, so that modification of addresses in branch instructions can easily be effected. We generate branch instructions with the aid of `jump(here, label)` and `jumponfalse(here, label)`, and we introduce two auxiliary routines `storelabel(location)` and `backpatch(location)` to remember the location of an instruction, and to be able to repair the address fields of incompletely generated branch instructions at a later stage. The code generator exports a special value of the `CGEN_labels` type that can be used to generate a temporary target destination for such incomplete instructions.

- We have so far made no mention of the forward reference tables which the reader may be dreading. In fact we can leave the system to sort these out implicitly, pointing to yet another advantage of the recursive descent method. A little thought should show that side-effects of allowing only the structured *WhileStatement* and *IfStatement* are that we never need explicit labels, and that we need the same number of implicit labels for each instance of any construct. These labels may be handled by declaring appropriate variables local to parsing routines like *IfStatement*; each time a recursive call is made to *IfStatement* new variables will come into existence, and remain there for as long as it takes to complete parsing of the construction, after which they will be discarded. When compiling an *IfStatement* we simply use a technique like the following (shown devoid of error handling for simplicity):

```
void IfStatement(void)
// IfStatement = "IF" Condition "THEN" Statement .
{ CGEN_labels testlabel;   // must be declared locally
  getsym();                // scan past IF
  Condition();             // generates code to evaluate Condition
  jumponfalse(testlabel,
              undefined);  // remember address of incomplete instruction
  accept(thensym);         // scan past THEN
  Statement();             // generates code for intervening Statement(s)
  backpatch(testlabel);    // use local test value stored by jumponfalse
}
```

If the interior call to `Statement` needs to parse a further *IfStatement*, another instance of `testlabel` will be created for the purpose. Clearly, all variables associated with handling implicit forward references must be declared "locally", or chaos will ensue.

- We may need to generate special housekeeping code as we enter or leave a *Block*. This may not be apparent in the case of a single block program - which is all our language allows at present - but will certainly be the case when we extend the language to support procedures. This code can be generated by the routines `openstackframe` (for code needed as we enter the program) and `leaveprogram` (for code needed as we leave it to return, perhaps, to the charge

of some underlying operating system).

- `gettop` is provided so that a source listing may give details of the object code addresses corresponding to statements in the source.

We are now in a position to show a fully attributed phrase structure grammar for a complete Clang compiler. This could be submitted to Coco/R to generate such a compiler, or could be used to assist in the completion of a hand-crafted compiler such as the one to be found on the source diskette. The power and usefulness of this notation should now be very apparent.

```
PRODUCTIONS
  Clang
  =                           (. TABLE_entries entry; .)
     "PROGRAM"
     Ident<entry.name>        (. debug = (strcmp(entry.name, "DEBUG") == 0);
                                 entry.idclass = TABLE_progs;
                                 Table->enter(entry); .)
     WEAK ";" Block "." .

  Block
  =                           (. int framesize = 0; .)
     SYNC { (  ConstDeclarations | VarDeclarations<framesize> )
     SYNC }                   (. /* reserve space for variables */
                                 CGen->openstackframe(framesize); .)
     CompoundStatement        (. CGen->leaveprogram();
                                 if (debug) /* demonstration purposes */
                                   Table->printtable(stdout); .) .

  ConstDeclarations
  =  "CONST" OneConst { OneConst } .

  OneConst
  =                           (. TABLE_entries entry; .)
     Ident<entry.name>        (. entry.idclass = TABLE_consts; .)
     WEAK "="
     Number<entry.c.value>    (. Table->enter(entry); .)
     ";" .

  VarDeclarations<int &framesize>
  =  "VAR" OneVar<framesize> { WEAK "," OneVar<framesize> } ";" .

  OneVar<int &framesize>
  =                           (. TABLE_entries entry; .)
                              (. entry.idclass = TABLE_vars;
                                 entry.v.size = 1; entry.v.scalar = true;
                                 entry.v.offset = framesize + 1; .)
     Ident<entry.name>
     [ UpperBound<entry.v.size> (. entry.v.scalar = false; .)
     ]                        (. Table->enter(entry);
                                 framesize += entry.v.size; .) .

  UpperBound<int &size>
  =  "[" Number<size> "]"     (. size++; .) .

  CompoundStatement
  =  "BEGIN" Statement { WEAK ";" Statement } "END" .

  Statement
  =  SYNC [  CompoundStatement | Assignment
           | IfStatement       | WhileStatement
           | ReadStatement     | WriteStatement
           | "STACKDUMP"      (. CGen->dump(); .)
         ] .

  Assignment
  =  Variable ":="
     Expression SYNC          (. CGen->assign(); .) .

  Variable
  =                           (. TABLE_entries entry; .)
     Designator<classset(TABLE_vars), entry> .

  Designator<classset allowed, TABLE_entries &entry>
  =                           (. TABLE_alfa name;
                                 bool found; .)
     Ident<name>              (. Table->search(name, entry, found);
```

```
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              if (entry.idclass != TABLE_vars) return;
                              CGen->stackaddress(entry.v.offset); .)
   ( "["                   (. if (entry.v.scalar) SemError(204); .)
     Expression            (. /* determine size for bounds check */
                              CGen->stackconstant(entry.v.size);
                              CGen->subscript(); .)
     "]"
     |                     (. if (!entry.v.scalar) SemError(205); .)
   ) .

IfStatement
=                          (. CGEN_labels testlabel; .)
   "IF" Condition "THEN"   (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Statement               (. CGen->backpatch(testlabel); .) .

WhileStatement
=                          (. CGEN_labels startloop, testlabel, dummylabel; .)
   "WHILE"                 (. CGen->storelabel(startloop); .)
   Condition "DO"          (. CGen->jumponfalse(testlabel, CGen->undefined); .)
   Statement               (. CGen->jump(dummylabel, startloop);
                              CGen->backpatch(testlabel); .) .

Condition
=                          (. CGEN_operators op; .)
   Expression
   (  RelOp<op> Expression (. CGen->comparison(op); .)
    | /* Missing op */     (. SynError(91); .)
   ) .

ReadStatement
=  "READ" "(" Variable     (. CGen->readvalue(); .)
   { WEAK "," Variable     (. CGen->readvalue(); .)
   } ")" .

WriteStatement
=  "WRITE" [ "(" WriteElement { WEAK "," WriteElement } ")" ]
                           (. CGen->newline(); .) .

WriteElement
=                          (. char str[600];
                              CGEN_labels startstring; .)
    String<str>            (. CGen->stackstring(str, startstring);
                              CGen->writestring(startstring); .)
  | Expression             (. CGen->writevalue(); .) .

Expression
=                          (. CGEN_operators op; .)
   (   "+" Term
     | "-" Term            (. CGen->negateinteger(); .)
     | Term
   )
   { AddOp<op> Term        (. CGen->binaryintegerop(op); .)
   } .

Term
=                          (. CGEN_operators op; .)
   Factor
   { (  MulOp<op>
      | /* missing op */   (. SynError(92); op = CGEN_opmul; .)
     ) Factor              (. CGen->binaryintegerop(op); .)
   } .

Factor
=                          (. TABLE_entries entry;
                              int value; .)
     Designator<classset(TABLE_consts, TABLE_vars), entry>
                           (. switch (entry.idclass)
                              { case TABLE_vars :
                                  CGen->dereference(); break;
                                case TABLE_consts :
                                  CGen->stackconstant(entry.c.value); break;
                              } .)
   | Number<value>         (. CGen->stackconstant(value); .)
   | "(" Expression ")" .

AddOp<CGEN_operators &op>
=    "+"                   (. op = CGEN_opadd; .)
   | "-"                   (. op = CGEN_opsub; .) .

MulOp<CGEN_operators &op>
=    "*"                   (. op = CGEN_opmul; .)
```

```
      | "/"                    (. op = CGEN_opdvd; .) .

    RelOp<CGEN_operators &op>
    =    "="                   (. op = CGEN_opeql; .)
       | "<>"                  (. op = CGEN_opneq; .)
       | "<"                   (. op = CGEN_oplss; .)
       | "<="                  (. op = CGEN_opleq; .)
       | ">"                   (. op = CGEN_opgtr; .)
       | ">="                  (. op = CGEN_opgeq; .) .

    Ident<char *name>
    =  identifier              (. LexName(name, TABLE_alfalength); .) .

    String<char *str>
    =  string                  (. char local[100];
                                  LexString(local, sizeof(local) - 1);
                                  int i = 0;
                                  while (local[i]) /* strip quotes */
                                  { local[i] = local[i+1]; i++; }
                                  local[i-2] = '\0';
                                  i = 0;
                                  while (local[i]) /* find internal quotes */
                                  { if (local[i] == '\'')
                                    { int j = i;
                                      while (local[j])
                                      { local[j] = local[j+1]; j++; }
                                    }
                                    i++;
                                  }
                                  strcpy(str, local); .) .

    Number <int &num>
    =  number                  (. char str[100];
                                  int i = 0, l, digit, overflow = 0;
                                  num = 0;
                                  LexString(str, sizeof(str) - 1);
                                  l = strlen(str);
                                  while (i <= l && isdigit(str[i]))
                                  { digit = str[i] - '0'; i++;
                                    if (num <= (maxint - digit) / 10)
                                      num = 10 * num + digit;
                                    else overflow = 1;
                                  }
                                  if (overflow) SemError(200); .) .

END Clang.
```

A few points call for additional comment:

- The reverse Polish (postfix) form of the expression manipulation is accomplished simply by delaying the calls for "operation" code generation until after the second "operand" code generation has taken place - this is, of course, completely analogous to the system developed in section 11.1 for converting infix expression strings to their reverse Polish equivalents.

- It turns out to be useful for debugging purposes, and for a full understanding of the way in which our machine works, to be able to print out the evaluation stack at any point in the program. This we have done by introducing another keyword into the language, `STACKDUMP`, which can appear as a simple statement, and whose code generation is handled by `dump`.

- The reader will recall that the production for *Factor* would be better expressed in a way that would introduce an LL(1) conflict into the grammar. This conflict is resolved within the production for *Designator* in the above Cocol grammar; it can be (and is) resolved within the production for *Factor* in the hand-crafted compiler on the source diskette. In other respects the semantic actions found in the hand-crafted code will be found to match those in the Cocol grammar very closely indeed.

Exercises

Many of the previous suggestions for extending Clang or Topsy will act as useful sources of inspiration for projects. Some of these may call for extra code generator interface routines, but many will be found to require no more than we have already discussed. Decide which of the following problems can be solved immediately, and for those that cannot, suggest the minimal extensions to the code generator that you can foresee might be necessary.

15.1 How do you generate code for the `REPEAT ... UNTIL` statement in Clang (or the `do` statement in Topsy)?

15.2 How do you generate code for an `IF ... THEN ... ELSE` statement, with the "dangling else" ambiguity resolved as in Pascal or C++? Bear in mind that the `ELSE` part may or may not be present, and ensure that your solution can handle both situations.

15.3 What sort of code generation is needed for the Pascal or Modula-2 style `FOR` loop that we have suggested adding to Clang? Make sure you understand the semantics of the `FOR` loop before you begin - they may be more subtle than you think!

15.4 What sort of code generation is needed for the C++ style `for` loop that we have suggested adding to Topsy?

15.5 Why do you suppose languages allow a `FOR` loop to terminate with its control variable "undefined"?

15.6 At present the `WRITE` statement of Clang is rather like Pascal's `WriteLn`. What changes would be needed to provide an explicit `WRITELN` statement, and, similarly, an explicit `READLN` statement, with semantics as used in Pascal.

15.7 If you add a "character" data type to your language, as suggested in Exercise 14.30, how do you generate code to handle `READ` and `WRITE` operations?

15.8 Code generation for the `LOOP ... EXIT ... END` construction suggested in Exercise 14.28 provides quite an interesting exercise. Since we may have several `EXIT` statements in a loop, we seem to have a severe forward reference problem. This may be avoided in several ways. For example, we could generate code of the form

```
           GOTO STARTLOOP
EXITPOINT  GOTO LOOPEXIT
STARTLOOP  code for loop body
           . . .
           GOTO EXITPOINT     (from an EXIT statement)
           . . .
           GOTO STARTLOOP
LOOPEXIT   code which follows loop
```

With this idea, all `EXIT` statements can branch back to `EXITPOINT`, and we have only to backpatch the one instruction at `EXITPOINT` when we reach the `END` of the `LOOP`. This is marginally inefficient, but the execution of one extra `GOTO` statement adds very little to the overall execution time.

Another idea is to generate code like

```
STARTLOOP  code for loop body
           . . .
           GOTO EXIT1    (from an EXIT statement)
           . . .
EXIT1      GOTO EXIT2    (from an EXIT statement)
```

```
            . . .
EXIT2       GOTO LOOPEXIT (from an EXIT statement)
            . . .
            GOTO STARTLOOP
LOOPEXIT    code which follows END
```

In this case, each time another `EXIT` is encountered the previously incomplete one is backpatched to branch to the incomplete instruction which is just about to be generated. When the `END` is encountered, the last one is backpatched to leave the loop. (A `LOOP ... END` structure may, unusually, have no `EXIT` statements, but this is easily handled.) This solution is even less efficient than the last. An ingenious modification can lead to much better code. Suppose we generate code which at first appears quite incorrect, on the lines of

```
STARTLOOP  code for loop body
           . . .
EXIT0      GOTO 0        (incomplete - from an EXIT statement)
           . . .
EXIT1      GOTO EXIT0    (from an EXIT statement)
           . . .
EXIT2      GOTO EXIT1    (from an EXIT statement)
           . . .
```

with an auxiliary variable `Exit` which contains the address of the most recent of the `GOTO` instructions so generated. (In the above example this would contain the address of the instruction labelled `EXIT2`). We have used only backward references so far, so no real problems arise. When we encounter the `END`, we refer to the instruction at `Exit`, alter its address field to the now known forward address, and use the old backward address to find the address of the next instruction to modify, repeating this process until the "`GOTO 0`" is encountered, which stops the chaining process - we are, of course, doing nothing other than constructing a linked list temporarily within the generated code.

Try out one or other approach, or come up with your own ideas. All of these schemes need careful thought when the possibility exists for having nested `LOOP ... END` structures, which you should allow.

15.9 What sort of code generation is needed for the translation of structured statements like the following?

```
IfStatement       =  "IF" Condition "THEN" StatementSequence
                       { "ELSIF" Condition "THEN" StatementSequence }
                       [ "ELSE" StatementSequence  ]
                     "END" .
WhileStatement    =  "WHILE" Condition "DO" StatementSequence  "END" .
StatementSequence =  Statement { ";" Statement } .
```

15.10 Brinch Hansen (1983) introduced an extended form of the `WHILE` loop into the language Edison:

```
WhileStatement    =  "WHILE" Condition "DO" StatementSequence
                       { "ELSE" Condition "DO" StatementSequence }
                     "END" .
```

The *Conditions* are evaluated one at a time in the order written until one is found to be true, when the corresponding *StatementSequence* is executed, after which the process is repeated. If no *Condition* is true, the loop terminates. How could this be implemented? Can you think of any algorithms where this statement would be useful?

15.11 Add a `HALT` statement, as a variation on the `WRITE` statement, which first prints the values of its parameters and then aborts execution.

15.12 How would you handle the `GOTO` statement, assuming you were to add it to the language?

What restrictions or precautions should you take when combining it with structured loops (and, in particular, `FOR` loops)?

15.13 How would you implement a `CASE` statement in Clang, or a `switch` statement in Topsy? What should be done to handle an `OTHERWISE` or `default`, and what action should be taken to be taken when the selector does not match any of the labelled "arms"? Is it preferable to regard this as an error, or as an implicit "do nothing"?

15.14 Add the `MOD` or `%` operator for use in finding remainders in expressions, and the `AND, OR` and `NOT` operations for use in forming more complex *Conditions*.

15.15 Add `INC(x)` and `DEC(x)` statements to Clang, or equivalently add `x++` and `x--` statements to Topsy - thereby turning it, at last, into Topsy++! The Topsy version will introduce an LL(1) conflict into the grammar, for now there will be three distinct alternatives for *Statement* that commence with an identifier. However, this conflict is not hard to resolve.

Further reading

The hypothetical stack machine has been widely used in the development of Pascal compilers. In the book by Welsh and McKeag (1980) can be found a treatment on which our own is partly based, as is the excellent treatment by Elder (1994). The discussion in the book by Wirth (1976b) is also relevant, although, as is typical in several systems like this, no real attempt is made to specify an interface to the code generation, which is simply overlaid directly onto the analyser in a machine dependent way. The discussion of the Pascal-P compiler in the book by Pemberton and Daniels (1982) is, as usual, extensive. However, code generation for a language supporting a variety of data types (something we have so far assiduously avoided introducing except in the exercises) tends to obscure many principles when it is simply layered onto an already large system.

Various approaches can be taken to compiling the `CASE` statement. The reader might like to consult the early papers by Sale (1981) and Hennessy and Mendelsohn (1982), as well as the descriptions in the book by Pemberton and Daniels (1982).

15.2 Code generation for a simple stack machine

The problem of code generation for a real machine is, in general, complex, and very specialized. In this section we shall content ourselves with completing our first level Clang compiler on the assumption that we wish to generate code for the stack machine described in section 4.4. Such a machine does not exist, but, as we saw, it may readily be emulated by a simple interpreter. Indeed, if the `interpret` routine from that section is invoked from the driver program after completing a successful parse, an implementation of Clang quite suitable for experimental work is readily produced.

An implementation of an "on-the-fly" code generator for this machine is almost trivially easy, and can be found on the source diskette. In studying this code the reader should note that:

- An external instance of the `STKMC` class is made directly visible to the code generator; as the code is generated it is stored directly in the code array `Machine->mem`.

- The constructor of the code generator class initializes two private members - a location counter (`codetop`) needed for storing instructions, and a top of memory pointer (`stktop`) needed for storing string literals.

- The `stackaddress` routine is passed a simple symbol table address value, and converts this into an offset that will later be computed relative to the `cpu.bp` register when the program is executed.

- The main part of the code generation is done in terms of calls to a routine `emit`, which does some error checking that the "memory" has not overflowed. Storing a string in the literal pool in high memory is done by routine `stackstring`, which is also responsible for overflow checking. As usual, errors are reported through the error reporting class discussed in section 14.3; the code generator suppresses further attempts to generate code if memory overflow occurs, while still allowing syntactic parsing to proceed.

- Since many of the routines in the code generator class are very elementary interfaces to `emit`, the reader might feel that we have taken modular decomposition too far - code generation as simple as this could surely be made more efficient if the parser simply evoked `emit` directly. This is certainly true, and many recursive descent compilers do this.

- Code generation is very easy for a stack-oriented language. It is much more difficult for a machine with no stack, and only a few registers and addressing modes. However, as the discussion in later sections will reveal, the interface we have developed is, in fact, capable of being used with only minor modification for code generation for more conventional machines. Developing the system in a highly modular way has many advantages if one is striving for portability, and for the ability to construct improved back ends easily.

It may be of interest to show the code generated for a simple program that incorporates several features of the language.

```
Clang 1.0 on 19/05/96 at 22:17:12

  0 : PROGRAM Debug;
  0 :   CONST
  0 :     VotingAge = 18;
  0 :   VAR
  0 :     Eligible, Voters[100], Age, Total;
  0 :   BEGIN
  2 :     Total := 0;
  7 :     Eligible := 0;
 12 :     READ(Age);
 15 :     WHILE Age > 0 DO
 23 :       BEGIN
 23 :         IF Age > VotingAge THEN
 29 :           BEGIN
 31 :             Voters[Eligible] := Age;
 43 :             Eligible := Eligible + 1;
 52 :             Total := Total + Voters[Eligible -1];
 67 :           END;
 71 :         READ(Age);
 74 :       END;
 76 :     WRITE(Eligible, ' voters. Average age = ', Total / Eligible);
 91 :   END.
```

The symbol table has entries

```
1 DEBUG          Program
2 VOTINGAGE      Constant      18
3 ELIGIBLE       Variable       1
4 VOTERS         Variable       2
5 AGE            Variable     103
6 TOTAL          Variable     104
```

and the generated code is as follows:

```
 0 DSP   104    Reserve variable space
 2 ADR  -104      address of Total
 4 LIT     0      push 0
 6 STO          Total := 0
 7 ADR    -1      address of Eligible
 9 LIT     0      push 0
11 STO          Eligible := 0
12 ADR  -103      address of Age
14 INN          READ(Age)
15 ADR  -103      address of Age
17 VAL            value of Age
18 LIT     0      push 0
20 GTR            compare
21 BZE    74    WHILE Age > 0 DO
23 ADR  -103      address of Age
25 VAL            value of Age
26 LIT    18      push VotingAge
28 GTR            compare
29 BZE    69      IF Age > VotingAge THEN
31 ADR    -2        address of Voters[0]
33 ADR    -1        address of Eligible
35 VAL              value of Eligible
36 LIT   101        array size 101
38 IND              address of Voters[Eligible]
39 ADR  -103        address of Age
41 VAL              value of Age
42 STO              Voters[Eligible] := Age
43 ADR    -1          address of Eligible
45 ADR    -1          address of Eligible
47 VAL                value of Eligible
48 LIT     1          push 1
50 ADD                value of Eligible + 1
51 STO              Eligible := Eligible + 1
52 ADR  -104          address of Total
54 ADR  -104          address of Total
56 VAL                value of Total
57 ADR    -2          address of Voters[0]
59 ADR    -1          address of Eligible
61 VAL                value of Eligible
62 LIT     1          push 1
64 SUB                value of Eligible - 1
65 LIT   101          array size 101
67 IND                address of Voters[Eligible-1]
68 VAL                value of Voters[Eligible - 1]
69 ADD                value of Total + Voters[Eligible - 1]
70 STO              Total := Total + Voters[Eligible - 1]
71 ADR  -103           address of Age
73 INN              READ(Age)
74 BRN    15        to start of WHILE loop
76 ADR    -1      address of Eligible
78 VAL            value of Eligible
79 PRN          WRITE(Eligible,
80 PRS  ' voters. Average age = '
82 ADR  -104      address of Total
84 VAL            value of Total
85 ADR    -1      address of Eligible
87 VAL            value of Eligible
88 DVD            value of Total / Eligible
89 PRN          WRITE(Total / Eligible)
90 NLN          output new line
91 HLT          END.
```

Exercises

15.16 If you study the code generated by the compiler you should be struck by the fact that the sequence `ADR x; VAL` occurs frequently. Investigate the possibilities for peephole optimization. Assume that the stack machine is extended to provide a new operation `PSH x` that will perform this sequence in one operation as follows:

```
case STKMC_psh:
  cpu.sp--;
  int ear = cpu.bp + mem[cpu.pc];
```

```
if (inbounds(cpu.sp) && inbounds(ear))
  { mem[cpu.sp] = mem[ear]; cpu.pc++; }
break;
```

Go on to modify the code generator so that it will replace any sequence `ADR x; VAL` with `PSH x` (be careful: not all `VAL` operations follow immediately on an `ADR` operation).

15.17 Augment the system so that you can declare constants to be literal strings and print these, for example

```
PROGRAM Debug;
  CONST
    Planet = 'World';
  BEGIN
    WRITE('Hello ', Planet)
  END.
```

How would you need to modify the parser, code generator, and run-time system?

15.18 Suppose that we wished to use relative branch instructions, rather than absolute branch instructions. How would code generation be affected?

15.19 (Harder) Once you have introduced a Boolean type into Clang or Topsy, along with `AND` and `OR` operations, try to generate code based on short-circuit semantics, rather than the easier Boolean operator approach. In the short-circuit approach the operators `AND` and `OR` are defined to have semantic meanings such that

```
A AND B       means      IF A THEN B ELSE FALSE END
A OR  B       means      IF A THEN TRUE ELSE B END
```

In the language Ada this has been made explicit: `AND` and `OR` alone have Boolean operator semantics, but `AND THEN` and `OR ELSE` have short-circuit semantics. Thus, in Ada

```
A AND THEN B       means      IF A THEN B ELSE FALSE END
A OR ELSE  B       means      IF A THEN TRUE ELSE B END
```

Can you make your system accept both forms?

15.20 Consider an extension where we allow a one-dimensional array with fixed bounds, but with the lower bound set by the user. For example, a way to declare such arrays might be along the lines of

```
CONST
  BC = -44;
  AD = 300;
VAR
  WWII[1939 : 1945], RomanBritain[BC : AD];
```

Modify the language, compiler, and interpreter to handle this idea, performing bounds checks on the subscript. Addressing an element is quite easy. If we declare an array

```
VAR Array[Min : Max];
```

then the offset of the *I* th element in the array is computed as

```
I - Min + offset of first element of array
```

which may give some hints about the checking problem too, if you think of it as

```
(offset of first element of array - Min)  +  I
```

15.21 A little more ingenuity is called for if one is to allow two-dimensional arrays. Again, if these

are of fixed size, addressing is quite easy. Suppose we declare a matrix

```
VAR Matrix[MinX : MaxX , MinY : MaxY];
```

Then we shall have to reserve `(MaxX-MinX+1) * (MaxY-MinY+1)` consecutive locations for the whole array. If we store the elements by rows (as in most languages, other than Fortran), then the offset of the *I,J* th element in the matrix will be found as

```
(I - MinX) * (MaxY - MinY + 1) + (J - MinY) + offset of first element
```

You will need a new version of the `STK_ind` opcode (incorporating bounds checking).

15.22 Extend your system to allow whole arrays (of the same length) to be assigned one to another.

15.23 Some users like to live dangerously. How could you arrange for the compiler to have an option whereby generation of subscript range checks could be suppressed?

15.24 Complete level 1 of your extended Clang or Topsy compiler by generating code for all the extra statement forms that you have introduced while assiduously exploring the exercises suggested earlier in this chapter. How many of these can only be completed if the instruction set of the machine is extended?

15.3 Other aspects of code generation

As the reader may have realized, the approach taken to code generation up until now has been rather idealistic. A hypothetical stack machine is, in many ways, ideal for our language - as witness the simplicity of the code generator - but it may differ rather markedly from a real machine. In this section we wish to look at other aspects of this large and intricate subject.

15.3.1 Machine tyranny

It is rather awkward to describe code generation for a real machine in a general text. It inevitably becomes machine specific, and the principles may become obscured behind a lot of detail for an architecture with which the reader may be completely unfamiliar. To illustrate a few of these difficulties, we shall consider some features of code generation for a relatively simple processor.

The Zilog Z80 processor that we shall use as a model is typical of several 8-bit microprocessors that were very popular in the early 1980's, and which helped to spur on the microcomputer revolution. It had a single 8-bit accumulator (denoted by `A`), several internal 8-bit registers (denoted by `B, C, D, E, H` and `L`), a 16-bit program counter (`PC`), two 16-bit index registers (`IX` and `IY`), a 16-bit stack pointer (`SP`), an 8-bit data bus, and a 16-bit address bus to allow access to 64KB of memory. With the exception of the `BRN` opcode (and, perhaps, the `HLT` opcode), our hypothetical machine instructions do not map one-for-one onto Z80 opcodes. Indeed, at first sight the Z80 would appear to be ill suited to supporting a high-level language at all, since operations on a single 8-bit accumulator only provide for handling numbers between -128 and +127, scarcely of much use in arithmetic calculations. For many years, however - even after the introduction of processors like the Intel 80x86 and Motorola 680x0 - 16-bit arithmetic was deemed adequate for quite a number of operations, as it allows for numbers in the range -32768 to +32767. In the Z80 a limited number of operations were allowed on 16-bit register pairs. These were denoted `BC, DE` and `HL`, and were formed by simply concatenating the 8-bit registers mentioned earlier. For example, 16-bit constants could be loaded immediately into a register pair, and such pairs could be pushed and popped from

the stack, and transferred directly to and from memory. In addition, the HL pair could be used as a 16-bit accumulator into which could be added and subtracted the other pairs, and could also be used to perform register-indirect addressing of bytes. On the Z80 the 16-bit operations stopped short of multiplication, division, logical operations and even comparison against zero, all of which are found on more modern 16 and 32-bit processors. We do not propose to describe the instruction set in any detail; hopefully the reader will be able to understand the code fragments below from the commentary given alongside.

As an example, let us consider Z80 code for the simple assignment statement

```
I := 4 + J - K
```

where I, J and K are integers, each stored in two bytes. A fairly optimal translation of this, making use of the HL register pair as a 16 bit accumulator, but not using a stack in any way, might be as follows:

```
LD    HL,4       ; HL := 4
LD    DE,(J)     ; DE := Mem[J]
ADD   HL,DE      ; HL := HL + DE          (4 + J)
LD    DE,(K)     ; DE := Mem[K]
OR    A          ; just to clear Carry
SBC   HL,DE      ; HL := HL - DE - Carry  (4 + J - K)
LD    (I),HL     ; Mem[I] := HL
```

On the Z80 this amounted to some 18 bytes of code. The only point worth noting is that, unlike addition, there was no simple 16-bit subtraction operation, only one which involved a carry bit, which consequently required unsetting before SBC could be executed. By contrast, the same statement coded for our hypothetical machine would have produced 13 words of code

```
ADR   I          ; push address of I
LIT   4          ; push constant 4
ADR   J          ; push address of J
VAL              ; replace with value of J
ADD              ; 4 + J
ADR   K          ; push address of K
VAL              ; replace with value of K
SUB              ; 4 + J - K
STO              ; store on I
```

and for a simple single-accumulator machine like that discussed in Chapter 4 we should probably think of coding this statement on the lines of

```
LDI   4          ; A := 4
ADD   J          ; A := 4 + J
SUB   K          ; A := 4 + J - K
STA   I          ; I := 4 + J - K
```

How do we begin to map stack machine code to these other forms? One approach might be to consider the effect of the opcodes, as defined in the interpreter in Chapter 4, and to arrange that code generating routines like stackaddress, stackconstant and assign generate code equivalent to that which would be obeyed by the interpreter. For convenience we quote the relevant equivalences again. (We use T to denote the virtual machine top of stack pointer, to avoid confusion with the SP register of the Z80 real machine.)

```
ADR address : T := T - 1; Mem[T] := address          (push an address)
LIT value   : T := T - 1; Mem[T] := value            (push a constant)
VAL         : Mem[T] := Mem[Mem[T]]                  (dereference)
ADD         : T := T + 1; Mem[T] := Mem[T] + Mem[T-1] (addition)
SUB         : T := T + 1; Mem[T] := Mem[T] - Mem[T-1] (subtraction)
STO         : Mem[Mem[T+1]] := Mem[T]; T := T + 2     (store top-of-stack)
```

It does not take much imagination to see that this would produce a great deal more code than we should like. For example, the equivalent Z80 code for an LIT opcode, obtained from a translation of

the sequence above, and generated by `stackconstant(Num)` might be

```
T := T - 1       :  LD    HL,(T)     ; HL := T
                    DEC   HL         ; HL := HL - 1
                    DEC   HL         ; HL := HL - 1
                    LD    (T),HL     ; T := HL
Mem[T] := Num    :  LD    DE,Num     ; DE := Num
                    LD    (HL),E     ; store low order byte
                    INC   HL         ; HL := HL + 1
                    LD    (HL),D     ; store high order byte
```

which amounts to some 14 bytes. We should comment that `HL` must be decremented twice to allow for the fact that memory is addressed in bytes, not words, and that we have to store the two halves of the register pair `DE` in two operations, "bumping" the `HL` pair (used for register indirect addressing) between these.

If the machine for which we are generating code does not have some sort of hardware stack we might be forced or tempted into taking this approach, but fortunately most modern processors do incorporate a stack. Although the Z80 did not support operations like `ADD` and `SUB` on elements of its stack, the pushing which is implicit in `LIT` and `ADR` is easily handled, and the popping and pushing implied by `ADD` and `SUB` are nearly as simple. Consequently, it would be quite simple to write code generating routines which, for the same assignment statement as before, would have the effects shown below.

```
ADR  I   :    LD    HL,I       ;     HL := address of I
              PUSH  HL         ; push address of I
LIT  4   :    LD    DE,4       ;     DE := 4
              PUSH  DE         ; push value of 4
ADR  J   :    LD    HL,J       ;     HL := address of J
              PUSH  HL         ; push address of J                  *
VAL      :    POP   HL         ;     HL := address of variable      *
              LD    E,(HL)     ;     E := Mem[HL]  low order byte
              INC   HL         ;     HL := HL + 1
              LD    D,(HL)     ;     D := Mem[HL]  high order byte
              PUSH  DE         ; replace with value of J            *
ADD      :    POP   DE         ;     DE := second operand           *
              POP   HL         ;     HL := first operand
              ADD   HL,DE      ;     HL := HL + DE
              PUSH  HL         ; 4 + J
ADR  K   :    LD    HL,K       ;     HL := address of K
              PUSH  HL         ; push address of K                  *
VAL      :    POP   HL         ;     HL := address of variable      *
              LD    E,(HL)     ;     E := low order byte
              INC   HL         ;     HL := HL + 1
              LD    D,(HL)     ;     D := high order byte
              PUSH  DE         ; replace with value of K            *
SUB      :    POP   DE         ;     DE := second operand           *
              POP   HL         ;     HL := first operand
              OR    A          ;     unset carry
              SBC   HL,DE      ;     HL := HL - DE - carry
              PUSH  HL         ; 4 + J - K                          **
STO      :    POP   DE         ;     DE := value to be stored       **
              POP   HL         ;     HL := address to be stored at
              LD    (HL),E     ;     Mem[HL] := E  store low order byte
              INC   HL         ;     HL := HL + 1
              LD    (HL),D     ;     Mem[HL] := D  store high order byte
                               ; store on I
```

We need not present code generator routines based on these ideas in any detail. Their intent should be fairly clear - the code generated by each follows distinct patterns, with obvious differences being handled by the parameters which have already been introduced.

For the example under discussion we have generated 41 bytes, which is still quite a long way from the optimal 18 given before. However, little effort would be required to reduce this to 32 bytes. It is easy to see that 8 bytes could simply be removed (the ones marked with a single asterisk), since the operations of pushing a register pair at the end of one code generating sequence and of popping the same pair at the start of the next are clearly redundant. Another byte could be removed by replacing

the two marked with a double asterisk by a one-byte opcode for exchanging the DE and HL pairs (the Z80 code EX DE,HL does this). These are examples of so-called "peephole" optimization, and are quite easily included into the code generating routines we are contemplating. For example, the algorithm for assign could be

```
PROCEDURE Assign;
(* Generate code to store top-of-stack on address stored next-to-top *)
  BEGIN
    IF last code generated was PUSH HL
      THEN replace this PUSH HL with EX DE,HL
      ELSIF last code generated was PUSH DE
        THEN delete PUSH DE
        ELSE generate code for POP DE
    END;
    generate code for POP HL; generate code for LD (HL),E
    generate code for INC HL; generate code for LD (HL),D
  END;
```

(The reader might like to reflect on the kinds of assignment statements which would give rise to the three possible paths through this routine.)

By now, hopefully, it will have dawned on the reader that generation of native code is probably strongly influenced by the desire to make this compact and efficient, and that achieving this objective will require the compiler writer to be highly conversant with details of the target machine, and with the possibilities afforded by its instruction set. We could pursue the ideas we have just introduced, but will refrain from doing so, concentrating instead on how one might come up with a better structure from which to generate code.

15.3.2 Abstract syntax trees as intermediate representations

In section 13.3 the reader was introduced to the idea of deriving an abstract syntax tree from an expression, by attributing the grammar that describes such expressions so as to add semantic actions that construct the nodes in these trees and then link them together as the parsing process is carried out. After such a tree has been completely constructed by the syntactic/semantic analysis phase (or even during its construction) it is often possible to carry out various transformations on it before embarking on the code generation phase that consists of walking it in some convenient way, generating code apposite to each node as it is visited.

In principle we can construct a single tree to represent an entire program. Initially we shall prefer simply to demonstrate the use of trees to represent the expressions that appear on both sides of *Assignments*, as components of *ReadStatements* and *WriteStatements* and as components of the *Condition* in *IfStatements* and *WhileStatements*. Later we shall extend the use of trees to handle the compilation of parameterized subroutine calls as well.

A tree is usually implemented as a structure in which the various nodes are linked by pointers, and so we declare an AST type to be a pointer to a NODE type. The nodes are inhomogeneous. When we declare them in traditional implementations we resort to using variant records or unions, but in a C++ implementation we can take advantage of inheritance to derive various node classes from a base class, declared as

```
struct NODE {
  int value;                    // value to be associated with this node
  bool defined;                 // true if value is predictable at compile time
  NODE()                        { defined = 0; }
  virtual void emit1(void) = 0;
  virtual void emit2(void) = 0;
  // ... further members as appropriate
};
```

where the emit member functions will be responsible for code generation as the nodes are visited

during the code generation phase. It makes sense to think of a value associated with each node - either a value that can be predicted at compile-time, or a value that will require code to be generated to compute it at run-time (where it will then be stored in a register, or on a machine stack, perhaps).

When constants form operands of expressions they give rise to nodes of a simple `CONSTNODE` class:

```
struct CONSTNODE : public NODE {
  CONSTNODE(int V)              { value = V; defined = true; }
  virtual void emit1(void);     // generate code to retrieve value of constant
  virtual void emit2(void)      {;}
};
```

Operands in expressions that are known to be associated with variables are handled by introducing a derived `VARNODE` class. Such nodes need to store the variable's offset address, and to provide at least two code generating member functions. These will handle the generation of code when the variable is associated with a *ValueDesignator* (as it is in a *Factor*) and when it is associated with a *VariableDesignator* (as it is on the left side of an *Assignment*, or in a *ReadStatement*).

```
struct VARNODE : public NODE {
  int offset;                   // offset of variable assigned by compiler
  VARNODE() {;}                 // default constructor
  VARNODE(int O)                { offset = O; }
  virtual void emit1(void);     // generate code to retrieve value of variable
  virtual void emit2(void);     // generate code to retrieve address of variable
};
```

To handle access to the elements of an array we derive an `INDEXNODE` class from the `VARNODE` class. The member function responsible for retrieving the address of an array element has the responsibility of generating code to perform run-time checks that the index remains in bounds, so we need further pointers to the subtrees that represent the index expression and the size of the array.

```
struct INDEXNODE : public VARNODE {
  AST size;                     // for range checking
  AST index;                    // subscripting expression
  INDEXNODE(int O, AST S, AST I) { offset = O; size = S; index = I; }
  //      void emit1(void)         is inherited from VARNODE
  virtual void emit2(void);     // code to retrieve address of array element
};
```

Finally, we derive two node classes associated with unary (prefix) and binary (infix) arithmetic operations

```
struct MONOPNODE : public NODE {
  CGEN_operators op;
  AST operand;
  MONOPNODE(CGEN_operators O, AST E) { op = O; operand = E }
  virtual void emit1(void);     // generate code to evaluate "op operand"
  virtual void emit2(void)      {;}
};

struct BINOPNODE : public NODE {
  CGEN_operators op;
  AST left, right;
  BINOPNODE(CGEN_operators O, AST L, AST R) { op = O; left = L; right = R; }
  virtual void emit1(void);     // generate code to evaluate "left op right"
  virtual void emit2(void)      {;}
};
```

The structures we hope to set up are exemplified by considering an assignment statement

```
  A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y
```

We use one tree to represent the address used for the destination (left side), and one for the value of the expression (right side), as shown in Figure 15.1, where for illustration the array `A` is assumed to have been declared with an size of 8.

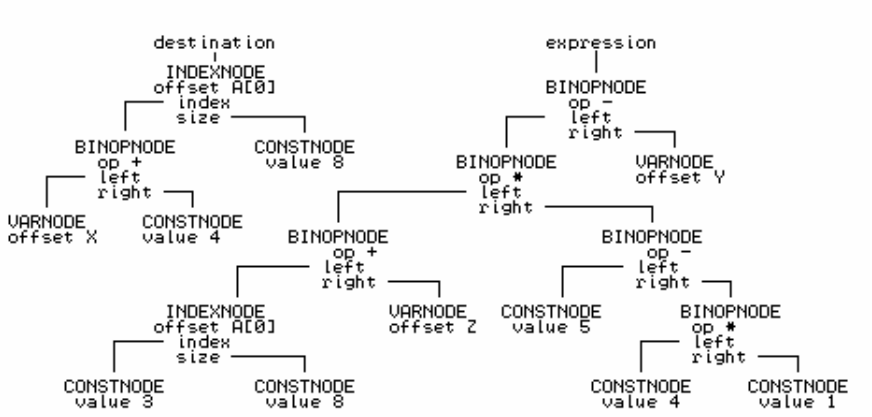


Figure 15.1 AST structures for the statement
A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y

Tree-building operations may be understood by referring to the attributes with which a Cocol grammar would be decorated:

```
Assignment
=                           (. AST dest, exp; .)
   Variable<dest> ":="
   Expression<exp> SYNC     (. CGen->assign(dest, exp); .) .

Variable<AST &V>
=                           (. TABLE_entries entry; .)
   Designator<V, classset(TABLE_vars), entry>.

Designator<AST &D, classset allowed, TABLE_entries &entry>
=                           (. TABLE_alfa name;
                               AST index, size;
                               bool found;
                               D = CGen->emptyast(); .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (!allowed.memb(entry.idclass)) SemError(206);
                               if (entry.idclass != TABLE_vars) return;
                               CGen->stackaddress(D, entry.v.offset); .)
   ( "["                    (. if (entry.v.scalar) SemError(204); .)
     Expression<index>      (. if (!entry.v.scalar)
                               /* determine size for bounds check */
                               { CGen->stackconstant(size, entry.v.size);
                                 CGen->subscript(D, entry.v.offset,
                                                 size, index); } .)
     "]"
     |                      (. if (!entry.v.scalar) SemError(205); .)
   ) .

Expression<AST &E>
=                           (. AST T;
                               CGEN_operators op;
                               E = CGen->emptyast(); .)
   (   "+" Term<E>
     | "-" Term<E>          (. CGen->negateinteger(E); .)
     | Term<E>
   )
   { AddOp<op> Term<T>      (. CGen->binaryintegerop(op, E, T); .)
   } .

Term<AST &T>
=                           (. AST F;
                               CGEN_operators op; .)
   Factor<T>
   { (  MulOp<op>
      | /* missing op */    (. SynError(92); op = CGEN_opmul; .)
     ) Factor<F>            (. CGen->binaryintegerop(CGEN_op, T, F); .)
   } .

Factor<AST &F>
=                           (. TABLE_entries entry;
                               int value;
                               F = CGen->emptyast(); .)
     Designator<F, classset(TABLE_consts, TABLE_vars), entry>
```

```
                          (. switch (entry.idclass)
                             { case TABLE_consts :
                                 CGen->stackconstant(F, entry.c.value);
                                 break;
                               default : break;
                             } .)
    | Number<value>         (. CGen->stackconstant(F, value); .)
    | "(" Expression<F> ")" .
```

The reader should note that:

- This grammar is very close to that presented earlier. The code generator interface is changed only in that the various routines need extra parameters specifying the subtrees that they manipulate.
- The productions for *Designator*, *Expression* and *Factor* take the precaution of initializing their formal parameter to point to an "empty" node, so that if a syntax error is detected, the nodes of a tree will still be well defined.

In a simple system, the various routines like `stackconstant, stackaddress` and `binaryintegerop` do little more than call upon the appropriate class constructors. As an example, the routine for `binaryintegerop` is merely

```
void binaryintegerop(CGEN_operators op, AST &left, AST &right)
{ left = new BINOPNODE(op, left, right); }
```

where we note that the `left` parameter is used both for input and output (this is done to keep the code generation interface as close as possible to that used in the previous system). These routines simply build the tree, and do not actually generate code.

Code generation is left in the charge of routines like `assign`, `jumponfalse` and `readvalue`, which take new parameters denoting the tree structures that they are required to walk. This may be exemplified by code for the `assign` routine, as it would be developed to generate code for our simple stack machine

```
void CGEN::assign(AST dest, AST expr)
{ if (dest)                // beware of corrupt trees
  { dest->emit2();         // generate code to push address of destination
    delete dest;           // recovery memory used for tree
  }
  if (expr)                // beware of corrupt trees
  { expr->emit1();         // generate code to push value of expression
    delete expr;           // recovery memory used for tree
    emit(int(STKMC_sto));  // generate the store instruction
  }
}
```

In typical OOP fashion, each subtree "knows" how to generate its own code! For a `VARNODE`, for example, and for our stack machine, we would define the `emit` members as follows:

```
void VARNODE::emit1(void)   // load variable value onto stack
{ emit2(); CGen->emit(int(STKMC_val)); }

void VARNODE::emit2(void)   // load variable address onto stack
{ CGen->emit(int(STKMC_adr)); CGen->emit(-offset); }
```

15.3.3 Simple optimizations - constant folding

The reader may need to be convinced that the construction of a tree is of any real value, especially when used to generate code for a simple stack machine. To back up the assertion that transformations on a tree are easily effected and can lead to the generation of better code, let us reconsider the statement

```
A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y
```

It is easy to identify opportunities for code improvement:

- `A[3]` represents an array access with a constant index. There is no real need to compute the additional offset for `A[3]` at run-time. It can be done at compile-time, along with a compile-time (rather than run-time) check that the subscript expression is "in bounds".

- Similarly, the subexpression (5 - 4 * 1) only has constant operands, and can also be evaluated at compile-time.

Before any code is generated, the trees for the above assignment could be reduced to those shown in Figure 15.2.

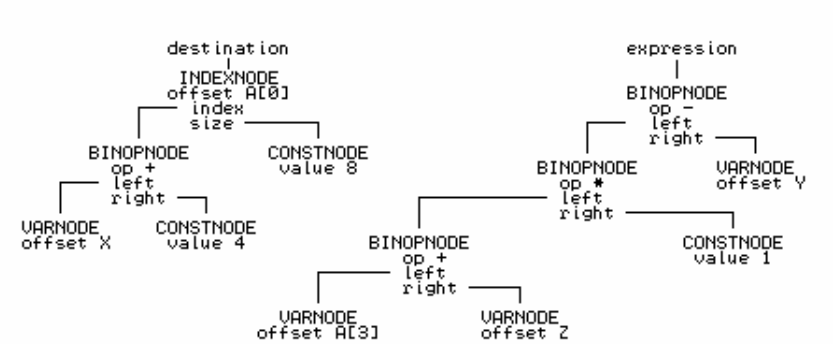


Figure 15.2 Improved AST structures for the statement A[X + 4] := (A[3] + Z) * (5 - 4 * 1) - Y

These sorts of manipulations fall into the category known as *constant folding*. They are easily added to the tree- building process, but are rather harder to do if code is generated on the fly. Constant folding is implemented by writing tree-building routines modelled on the following:

```
void CGEN::subscript(AST &base, int offset, AST &size, AST &index)
{ if (!index || !index->defined           // check for well defined
      || !size || !size->defined)         // trees and constant index
  { base = new INDEXNODE(offset, size, index); return; }
  if (unsigned(index->value) >= size->value)
    Report->error(223);                   // report range error immediately
  else                                    // simple variable designator
    base = new VARNODE(offset + index->value);
  delete index; delete size;              // and delete the unused debris
}

void CGEN::binaryop(CGEN_operators op, AST &left, AST &right)
{ if (left && right)                      // beware of corrupt trees
  { if (left->defined && right->defined)  // both operands are constant
    { switch (op)                         // so do compile-time evaluation
      { case CGEN_opadd: left->value += right->value; break;
        case CGEN_opsub: left->value -= right->value; break;
                                          // ... others like this
      }
      delete right; return;               // discard one operand
    }
  }
  left = new BINOPNODE(op, left, right);  // construct proper bin op node
}
```

The reader should notice that such constant folding is essentially machine independent (assuming that the arithmetic can be done at compile-time to the required precision). Tree construction represents the last phase of a machine-independent front end to a compiler; the routines that walk the tree become machine dependent.

Recognition and evaluation of expressions in which every operand is constant is useful if one

wishes to extend the language in other ways. For example, we may now easily extend our Clang language to allow for constant expressions within *ConstDeclarations*:

```
ConstDeclarations =  "CONST" OneConst { OneConst } .
OneConst          =  identifier "=" ConstExpression ";" .
ConstExpression   =  Expression .
```

We can make use of the existing parsing routines to handle a *ConstExpression*. The attributes in a Cocol specification would simply incorporate a constraint check that the expression was, indeed, "defined", and if so, store the "value" in the symbol table.

15.3.4 Simple optimizations - removal of redundant code

Production quality compilers often expend considerable effort in the detection of structures for which no code need be generated at all. For example, a source statement of the form

```
WHILE TRUE DO Something
```

does not require the generation of code like

```
LAB  IF NOT TRUE GOTO EXIT END
     Something
     GOTO LAB
EXIT
```

but can be reduced to

```
LAB  Something
     GOTO LAB
```

and, to take a more extreme case, if it were ever written, source code like

```
WHILE 15 < 6 DO Something
```

could be disregarded completely. Once again, optimizations of this sort are most easily attempted after an internal representation of the source program has been created in the form of a tree or graph. A full discussion of this fascinating subject is beyond the scope of this text, and it will suffice merely to mention a few improvements that might be incorporated into a simple tree-walking code generator for expressions. For example, the remaining multiplication by 1 in the expression we have used for illustration is redundant, and is easily eliminated. Similarly, multiplications by small powers of 2 could be converted into shift operations if the machine supports these, and multiplication by zero could be recognized as a golden opportunity to load a constant of 0 rather than perform any multiplications at all. To exemplify this, consider an extract from an improved routine that generates code to load the value resulting from a binary operation onto the run-time stack of our simple machine:

```
void BINOPNODE::emit1(void)
// load value onto stack resulting from binary operation
{ bool folded = false;
  if (left && right)                      // beware of corrupt trees
  { switch (op)                           // redundant operations?
    { case CGEN_opadd:
        if (right->defined && right->value == 0) // x + 0 = x
          { left->emit1(); folded = true; }
                                          // ... other special cases
        break;
      case CGEN_opsub:
                                          // ... other special cases
      case CGEN_opmul:
        if (right->defined && right->value == 1) // x * 1 = x
          { left->emit1(); folded = true; }
        else if (right->defined && right->value == 0) // x * 0 = 0
          { right->emit1(); folded = true; }
                                          // ... other special cases
```

```
          break;
        case CGEN_opdvd:
                                          // ... other special cases
      }
    }
    if (!folded)                          // still have to generate code
    { if (left)  left->emit1();           // beware of corrupt trees
      if (right) right->emit1();
      CGen->emit(int(STKMC_add) + int(op));   // careful - ordering used
    }
    delete left; delete right;            // remove debris
  }
```

These sorts of optimizations can have a remarkable effect on the volume of code that is generated - assuming, of course, that the expressions are littered with constants.

So far we have assumed that the structures set up as we parse expressions are all binary trees - each node has subtrees that are disjoint. Other structures are possible, although creating these calls for routines more complex than we have considered up till now. If we relax the restriction that subtrees must be disjoint, we introduce the possibility of using a so-called **directed acyclic graph** (DAG). This finds application in optimizations in which common subexpressions are identified, so that code for them is generated as few times as possible. For example, the expression *(a * a + b * b) / (a * a - b * b)* could be optimized so as to compute each of *a * a* and *b * b* only once. A binary tree structure and a DAG for this expression are depicted in Figure 15.3, but further treatment of this topic is beyond the scope of this text.

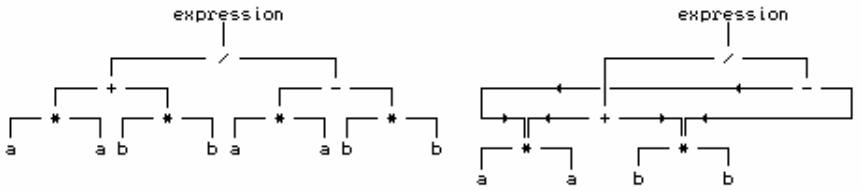


Figure 15.3 (a) Binary tree corresponding to expression
(a * a + b * b) / (a * a - b * b)
(b) DAG corresponding to expression
(a * a + b * b) / (a * a - b * b)

15.3.5 Generation of assembler code

We should also mention another approach often taken in providing native code compilers, especially on small machines. This is to generate output in the form of assembler code that can then be processed in a second pass using a macro assembler. Although carrying an overhead in terms of compilation speed, this approach has some strong points - it relieves the compiler writer of developing intensely machine dependent bit manipulating code (very tiresome in some languages, like the original Pascal), handling awkward forward referencing problems, dealing with operating system and linkage conventions, and so forth. It is widely used on Unix systems, for example.

On the source diskette can be found such a code generator. This can be used to construct a compiler that will translate Clang programs into the ASSEMBLER language for the tiny single-accumulator machine discussed in Chapter 4, and for which assemblers were developed in Chapter 6. Clearly there is a very real restriction on the size of source program that can be handled by this system, but the code generator employs several optimizations of the sort discussed earlier, and is an entertaining example of code that the reader is encouraged to study. Space does not permit of a full description, but the following points are worth emphasizing:

- An on-the-fly code generator for this machine would be very difficult to write, but the Cocol description of the phrase structure grammar can remain exactly the same as that used for the

stack machine. Naturally, the internal definitions of some members of the node classes are different, as are the implementations of the tree- walking member functions.

- The single-accumulator machine has conditional branching instructions that are very different from those used in the stack machine; it also has a rather non-orthogonal set of these. This calls for some ingenuity in the generation of code for *Conditions*, *IfStatements* and *WhileStatements*.

- The problem of handling the forward references needed in conditional statements is left to the later assembler stage. However, the code generator still has to solve the problem of generating a self-consistent set of labels for those instructions that need them.

- The input/output facilities of the two machines are rather disparate. In particular the single-accumulator machine does not have an special operation for writing strings. This is handled by arranging for the code generator to create and call a standard output subroutine for this purpose when it is required. The approach of generating calls to standardized library routines is, of course, very widespread in real compilers.

- Although capable of handling access to array elements, the code generator does not generate any run-time subscript checks, as these would be prohibitively expensive on such a tiny machine.

- The machine described in Chapter 4 does not have any operations for handling multiplication and division. A compiler error is reported if it appears that such operations are needed.

Exercises

Implementations of tree-based code generators for our simple stack machine can be found on the source diskette, as can the parsers and Cocol grammars that match these. The Modula-2 and Pascal implementations make use of variant records for discriminating between the various classes of nodes; C++ versions of these are also available.

15.25 If you program in Modula-2 or Pascal and have access to an implementation that supports OOP extensions to these languages, derive a tree-walking code generator based on the C++ model.

15.26 The constant folding operations perform little in the way of range checks. Improve them.

15.27 Adapt the tree-walking code generator for the stack machine to support the extensions you have made to Clang or Topsy.

15.28 Adapt the tree-walking code generator for the single-accumulator machine to support the extensions you have made to Clang or Topsy.

15.29 Extend the single-accumulator machine to support multiplication and division, and extend the code generator for Clang or Topsy to permit these operations (one cannot do much multiplication and division in an 8-bit machine, but it is the principle that matters here).

15.30 Follow up the suggestion made earlier, and extend Clang or Topsy to allow constant expressions to appear in constant declarations, for example

```
CONST
  Max = 100;
  Limit = 2 * Max + 1;
  NegMax = - Max;
```

15.31 Perusal of our example assignment should suggest the possibility of producing a still smaller tree for the right-hand side expression (Figure 15.4(a)). And, were the assignment to have been

```
A[X + 4] := (A[3] + Z) * (4 - 4 * 1) - Y
```

perhaps we could do better still (see Figure 15.4(b)). How would you modify the tree-building routines to achieve this sort of improvement? Can you do this in a way that still allows your compiler to support the notion of a constant expression as part of a constant declaration?

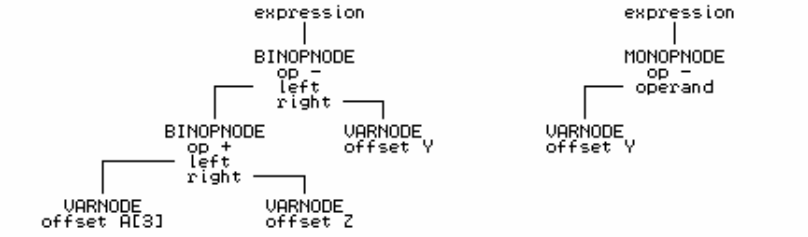


Figure 15.4 Optimal AST structures for the expressions
(a) (A[3] + Z) * (5 - 4 * 1) - Y (b) (A[3] + Z) * (4 - 4 * 1) - Y

15.32 (More extensive) Modify the attributed grammar and tree-building code generator so that node classes are introduced for the various categories of *Statement*. Then develop code generator routines that can effect the sorts of optimizations hinted at earlier for removing redundant code for unreachable components of *IfStatements* and *WhileStatements*. Sophisticated compilers often issue warnings when they discover code that can never be executed. Can you incorporate such a feature into your compiler?

15.33 (Harder) Use a tree-based representation to generate code for Boolean expressions that require short- circuit semantics (see Exercises 13.10 and 15.19).

15.34 (More extensive) Develop a code generator for a register-based machine such as that suggested in section 13.3. Can you do this without altering the Cocol specification, as we claim is possible for the single-accumulator machine of Chapter 4?

15.35 Many development environments incorporate "debuggers" - sophisticated tools that will trace the execution of a compiled program in conjunction with the source code, referring run-time errors to source code statements, allowing the user to interrogate (and even alter) the values of variables by using the identifiers of the source code, and so on. Development of such a system could be a very open- ended project. As a less ambitious project, extend the interpretive compiler for Clang or Topsy that, in the event of a run-time error, will relate this to the corresponding line in the source, and then print a post-mortem dump showing the values of the variables at the time the error occurred. A system of this sort was described for the well-known subset of Pascal known as Pascal-S (Wirth, 1981; Rees and Robson, 1987), and is also used in the implementation of the simple teaching language Umbriel (Terry, 1995).

15.36 Develop a code generator that produces correct C or C++ code from Clang or Topsy source.

Further reading

Our treatment of code generation has been dangerously superficial. "Real" code generation tends to become highly machine dependent, and the literature reflects this. Although all of the standard texts have a lot to say on the subject, those texts which do not confine themselves to generalities (by stopping short of showing how it is actually done) inevitably relate their material to one or other real machine, which can become confusing for a beginner who has little if any experience of that machine. Watson (1989) has a very readable discussion of the use of tree structures. Considerably more detail is given in the comprehensive books by Aho, Sethi and Ullman (1986) and Fischer and LeBlanc (1988, 1991). Various texts discuss code generation for familiar microprocessors. For example, the book by Mak (1991) develops a Pascal compiler that generates assembler code for the Intel 80x86 range of machines, and the book by Ullman (1994) develops a subset Modula-2 compiler that generates a variant of Intel assembler. The recent book by Holmes (1995) uses object orientation to develop a Pascal compiler, discussing the generation of assembler code for a SUN SPARC workstation. Wirth (1996) presents a tightly written account of developing a compiler for a subset of Oberon that generates code for a slightly idealized processor, modelled on the hypothetical RISC processor named DLX by Hennessy and Patterson (1990) to resemble the MIPS processor. Elder (1994) gives a thorough description of many aspects of code generation for a more advanced stack-based machine than the one described here.

16 SIMPLE BLOCK STRUCTURE

Our simple language has so far not provided for the *procedure* concept in any way. It is the aim of the next two chapters to show how Clang and its compiler can be extended to provide procedures and functions, using a model based on those found in block-structured languages like Modula-2 and Pascal, which allow the use of local variables, local procedures and recursion. This involves a much deeper treatment of the concepts of storage allocation and management than we have needed previously.

As in the last two chapters, we shall develop our arguments by a process of slow refinement. On the source diskette will be found Cocol grammars, hand-crafted parsers and code generators covering each stage of this refinement, and the reader is encouraged to study this code in detail as he or she reads the text.

16.1 Parameterless procedures

In this chapter we shall confine discussion to parameterless **regular procedures** (or *void functions* in C++ terminology), and discuss parameters and value-returning functions in the following chapter.

16.1.1 Source handling, lexical analysis and error reporting

The extensions to be discussed in this chapter require no changes to the source handler, scanner or error reporter classes that were not pre-empted in the discussion in Chapter 14.

16.1.2 Syntax

Regular procedure declaration is inspired by the way it is done in Modula-2, described in EBNF by

```
Block            =   { ConstDeclarations | VarDeclarations | ProcDeclaration }
                     CompoundStatement .
ProcDeclaration  =  "PROCEDURE" ProcIdentifier ";" Block ";" .
```

It might be thought that the same effect could be achieved with

```
ProcDeclaration  =  "PROCEDURE" ProcIdentifier ";" CompoundStatement ";" .
```

but the syntax first suggested allows for nested procedures, and for named constants and variables to be declared local to procedures, in a manner familiar to all Modula-2 and Pascal programmers.

The declaration of a procedure is most easily understood as a process whereby a *CompoundStatement* is given a name. Quoting this name at later places in the program then implies execution of that *CompoundStatement*. By analogy with most modern languages we should like to extend our definition of *Statement* as follows:

```
Statement        =   [  CompoundStatement | Assignment | ProcedureCall
                      | IfStatement | WhileStatement
                      | WriteStatement | ReadStatement ] .
ProcedureCall    =  ProcIdentifier .
```

However, this introduces a non-LL(1) feature into the grammar, for now we have two alternatives for *Statement* (namely *Assignment* and *ProcedureCall*) that begin with lexically indistinguishable

symbols. There are various ways of handling this problem:

- A purely syntactic solution for this simple language is possible if we re-factor the grammar as

```
Statement         =   [  CompoundStatement | AssignmentOrCall
                       | IfStatement | WhileStatement
                       | WriteStatement | ReadStatement ] .
AssignmentOrCall  =  Designator [ ":=" Expression ] .
```

  so that a *ProcedureCall* can be distinguished from an *Assignment* by simply looking at the first symbol after the *Designator*. This is, of course, the approach that has to be followed when using Coco/R.

- A simple solution would be to add the keyword `CALL` before a procedure identifier, as in Fortran, but this rather detracts from readability of the source program.

- Probably because the semantics of procedure calls and of assignments are so different, the solution usually adopted in a hand-crafted compiler is a static semantic one. To the list of allowed classes of identifier we add one that distinguishes procedure names uniquely. When the symbol starting a *Statement* is an identifier we can then determine from its symbol table attributes whether an *Assignment* or *ProcedureCall* is to be parsed (assuming all identifiers to have been declared before use, as we have been doing).

16.1.3 The scope and extent of identifiers

Allowing nested procedures - or even local variables on their own - introduces the concept of **scope**, which should be familiar to readers used to block-structured languages, although it often causes confusion to many beginners. In such languages, the "visibility" or "accessibility" of an identifier declared in a *Block* is limited to that block, and to blocks themselves declared local to that block, with the exception that when an identifier is *redeclared* in one or more nested blocks, the innermost accessible declaration applies to each particular use of that identifier.

Perhaps any confusion which arises in beginners' minds is exacerbated by the fact that the rather fine distinction between compile-time and run-time aspects of scope is not always made clear. At compile-time, only those names that are currently "in scope" will be recognized when translating statements and expressions. At run-time, each variable has an **extent** or **lifetime**. Other than the "global variables" (declared within the main program in Modula-2 or Pascal, or outside of all functions in C++), the only variables that "exist" at any one instant (that is, have storage allocated to them, with associated values) are those that were declared local to the blocks that are "active" (that is, are associated with procedures that have been called, but which have not yet returned).

One consequence of this, which a few readers may have fallen foul of at some stage, is that variables declared local to a procedure cannot be expected to retain their values between calls on the procedure. This leads to a programming style where many variables are declared globally, when they should, ideally, be "out of scope" to many of the procedures in the program. (Of course, the use of modules (in languages like Modula-2) or classes (in C++) allows many of these to be hidden safely away.)

Exercises

16.1 Extend the grammar for Topsy so as to support a program model more like that in C and C++,

in which routines may not be nested, although both global and local variables (and constants) may be declared.

16.1.4 Symbol table support for the scope rules

Scope rules like those suggested in the last section may be easily handled in a number of ways, all of which rely on some sort of stack structure. The simplest approach is to build the entire symbol table as a stack, pushing a node onto this stack each time an identifier is declared, and popping several nodes off again whenever we complete parsing a *Block*, thereby ensuring that the names declared local to that block then go out of scope. The stack structure also ensures that if two identifiers with the same names are declared in nested blocks, the first to be found when searching the table will be the most recently declared. The stack of identifier entries must be augmented in some way to keep track of the divisions between procedures, either by introducing an extra variant into the possibilities for the `TABLE_entries` structure, or by constructing an ancillary stack of special purpose nodes.

The discussion will be clarified by considering the shell of a simple program:

```
PROGRAM Main;
  VAR G1;                   (* global *)

  PROCEDURE One;
    VAR L1, L2;             (* local to One *)
    BEGIN
      (* body of One *)
    END;

  BEGIN
    (* body of Main *)
  END.
```

For this program, either of the approaches suggested by Figure 16.1(a) or (b) would appear to be suitable for constructing a symbol table. In these structures, an extra "sentinel" node has been inserted at the bottom of the stack. This allows a search of the table to be implemented as simply as possible, by inserting a copy of the identifier that is being sought into this node before the (linear) search begins.

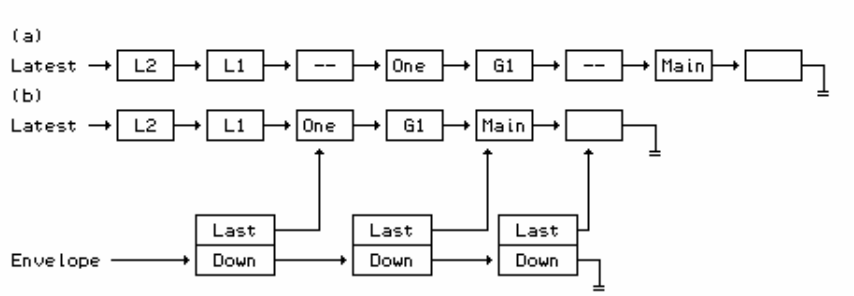


Figure 16.1 Stack based Symbol Table (a) with extra nodes marking scope boundaries (b) with ancillary stack marking scope boundaries

As it happens, this sort of structure becomes rather more difficult to adapt when one extends the language to allow procedures to handle parameters, and so we shall promote the idea of having a stack of *scope nodes*, each of which contains a pointer to the scope node corresponding to an outer scope, as well as a pointer to a structure of *identifier nodes* pertinent to its own scope. This latter structure could be held as a stack, queue, tree, or even hash table. Figure 16.2 shows a situation where queues have been used, each of which is terminated by a common sentinel node.

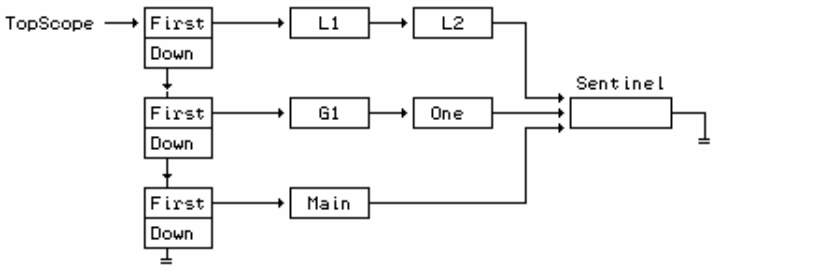


Figure 16.2 Symbol Table based on a set of queues, with ancillary stack marking scope boundaries

Although it may not immediately be seen as necessary, it turns out that to handle the addressing aspects needed for code generation we shall need to associate with each identifier the *static level* at which it was declared. The revised public interface to the symbol table class requires declarations like

```
enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs, TABLE_procs };

struct TABLE_entries {
  TABLE_alfa name;             // identifier
  int level;                   // static level
  TABLE_idclasses idclass;     // class
  union {
    struct {
      int value;
    } c;                       // constants
    struct {
      int size, offset;
      bool scalar;
    } v;                       // variables
    struct {
      CGEN_labels entrypoint;
    } p;                       // procedures
  };
};

class TABLE {
  public:
    void openscope(void);
    // Opens new scope before parsing a block

    void closescope(void);
    // Closes scope after parsing a block

    // rest as before (see section 14.6.3)
};
```

On the source diskette can be found implementations of this symbol table handler, while a version extended to meet the requirements of Chapter 17 can be found in Appendix B. As usual, a few comments on implementation techniques may be helpful to the reader:

- The symbol table handler manages the computation of `level` internally.
- An `entry` is passed by reference to the `enter` routine, so that, when required, the caller is able to retrieve this value after an entry has been made.
- The outermost program block can be defined as level 1 (some authors take it as level 0, others reserve this level for standard "pervasive" identifiers - like `INTEGER, BOOLEAN, TRUE` and `FALSE`).
- It is possible to have more than one entry in the table with the same `name`, although not within a single scope. The routine for adding an entry to the table checks that this constraint is

obeyed. However, a second occurrence of an identifier in a single scope will result in a further entry in the table.

- The routine for searching the symbol table works its way through the various scope levels from innermost to outermost, and is thus more complex than before. A call to `search` will, however, always return with a value for `entry` which matches the `name`, even if this had not been correctly declared previously. Such undeclared identifiers will seem to have an effective `idclass = TABLE_progs`, which will always be semantically unacceptable when the identifier is analysed further.

Exercises

16.2 Follow up the suggestion that the symbol table can be stored in a stack, using one or other of the methods suggested in Figure 16.1.

16.3 Rather than use a separate `SCOPE_nodes` structure, develop a version of the symbol table class that simply introduces another variant into the existing `TABLE_entries` structure, that is, extend the enumeration to

```
enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs, TABLE_procs,
TABLE_scopes };
```

16.4 How, if at all, does the symbol table interface require modification if you wish to develop the Topsy language to support `void` functions?

16.5 In our implementation of the table class, scope nodes are deleted by the `closescope` routine. Is it possible or advisable also to delete identifier nodes when identifiers go out of scope?

16.6 Some compilers make use of the idea of a forest of binary search trees. Develop a table handler to make use of this approach. How do you adapt the idea that a call to `search` will always return a well-defined `entry`?

For example, given source code like

```
PROGRAM Silly;
  VAR B, A, C;

  PROCEDURE One;
    VAR X, Y, Z;

    PROCEDURE Two;
      VAR Y, D;
```

the symbol table might look like that shown in Figure 16.3 immediately after declaring `D`.

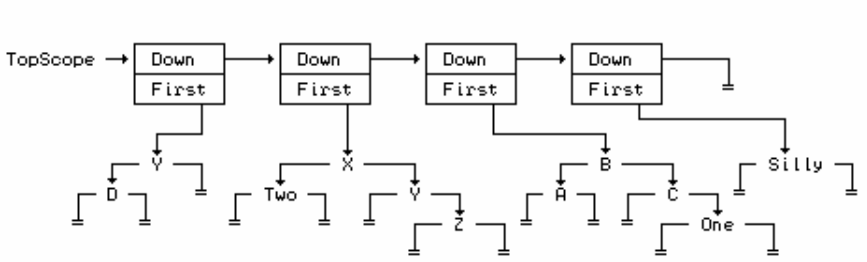


Figure 16.3 Symbol Table based on a forest of Binary Search Trees

---

Further reading

More sophisticated approach to symbol table construction are discussed in many texts on compiler construction. A very readable treatment may be found in the book by Elder (1994), who also discusses the problems of using a hash table approach for block-structured languages.

16.1.5 Parsing declarations and procedure calls

The extensions needed to the attributed grammar to describe the process of parsing procedure declarations and calls can be understood with reference to the Cocol extract below, where, for temporary clarity, we have omitted the actions needed for code generation, while retaining those needed for constraint analysis:

```
PRODUCTIONS
  Clang
  =                          (. TABLE_entries entry; .)
     "PROGRAM"
     Ident<entry.name>       (. entry.idclass = TABLE_progs;
                                Table->enter(entry); Table->openscope(); .)
     WEAK ";" Block "." .

  Block
  =  SYNC
     { ( ConstDeclarations | VarDeclarations | ProcDeclaration ) SYNC }
     CompoundStatement       (. Table->closescope(); .) .

  ProcDeclaration
  =                          (. TABLE_entries entry; .)
     "PROCEDURE"
     Ident<entry.name>       (. entry.idclass = TABLE_procs;
                                Table->enter(entry); Table->openscope(); .)
     WEAK ";"  Block ";" .

  Statement
  =  SYNC [  CompoundStatement | AssignmentOrCall | IfStatement
           | WhileStatement | ReadStatement | WriteStatement ] .

  AssignmentOrCall
  =                          (. TABLE_entries entry; .)
     Designator<classset(TABLE_vars, TABLE_procs), entry>
     (                       (. if (entry.idclass != TABLE_vars) SemError(210); .)
        ":=" Expression SYNC
      |                      (. if (entry.idclass != TABLE_procs) SemError(210); .)
     ) .
```

The reader should note that:

- Variables declared local to a *Block* will be associated with a level one higher than the block identifier itself.

- In a hand-crafted parser we can resolve the LL(1) conflict between the assignments and procedure calls within the parser for *Statement*, on the lines of

```
void Statement(symset followers)
{ TABLE_entries entry; bool found;
  if (FirstStatement.memb(SYM.sym))            // allow for empty statements
  { switch (SYM.sym)
    { case SCAN_identifier:                    // must resolve LL(1) conflict
        Table->search(SYM.name, entry, found);  // look it up
        if (!found) Report->error(202);         // undeclared identifier
        if (entry.idclass == TABLE_procs) ProcedureCall(followers, entry);
        else Assignment(followers, entry);
        break;
      case SCAN_ifsym:                         // other statement forms
        IfStatement(followers); break;         // as needed
  }
  test(followers, EmptySet, 32);               // synchronize if necessary
}
```

Exercises

16.7 In Exercise 14.50 we suggested that undeclared identifiers might be entered into the symbol table (and assumed to be variables) at the point where they were first encountered. Investigate whether one can do better than this by examining the symbol which appears after the offending identifier.

16.8 In a hand-crafted parser, when calling *Block* from within *ProcDeclaration* the semicolon symbol has to be added to *Followers*, as it becomes the legal follower of *Block*. Is it a good idea to do this, since the semicolon (a widely used and abused token) will then be an element of all *Followers* used in parsing parts of that block? If not, what does one do about it?

16.2 Storage management

If we wish procedures to be able to call one another recursively, we shall have to think carefully about code generation and storage management. At run-time there may at some stage be several *instances* of a recursive procedure in existence, pending completion. For each of these the corresponding instances of any local variables must be distinct. This has a rather complicating effect at compile-time, for a compiler can no longer associate a simple address with each variable as it is declared (except, perhaps, for the global variables in the main program block). Other aspects of code generation are not quite such a problem, although we must be on our guard always to generate so-called *re-entrant code*, which executes without ever modifying itself.

16.2.1 The stack frame concept

Just as the stack concept turns out to be useful for dealing with the compile-time accessibility aspects of *scope* in block-structured languages, so too do stack structures provide a solution for dealing with the run-time aspects of *extent* or *existence*. Each time a procedure is called, it acquires a region of free store for its local variables - an area which can later be freed when control returns to the caller. On a stack machine this becomes almost trivially easy to arrange, although it may be more obtuse on other architectures. Since procedure activations strictly obey a first-in-last-out scheme, the areas needed for their local working store can be carved out of a single large stack. Such areas are usually called **activation records** or **stack frames**, and do not themselves contain any code. In each of them is usually stored some standard information. This includes the **return address** through which control will eventually pass back to the calling procedure, as well as information that can later be used to reclaim the frame storage when it is no longer required. This

housekeeping section of the frame is called the **frame header** or **linkage area**. Besides the storage needed for the frame header, space must be also be allocated for local variables (and, possibly, parameters, as we shall see in a later section).

This may be made clearer by a simple example. Suppose we come up with the following variation on code for satisfying the irresistible urge to read a list of numbers and write it down in reverse order:

```
PROGRAM Backwards;
  VAR Terminator;

  PROCEDURE Start;
    VAR Local1, Local2;

    PROCEDURE Reverse;
      VAR Number;
      BEGIN
        Read(Number);
        IF Terminator <> Number THEN Start;  10: Write(Number)
      END;

    BEGIN (* Start *)
      Reverse;  20:
    END;

  BEGIN (* Backwards *)
    Terminator := 9;
    Start; 30:
  END (* Backwards *).
```

(Our language does not provide for labels; these have simply been added to make the descriptions easier.)

We note that a stack is also the obvious structure to use in a non-recursive solution to the problem, so the example also highlights the connection between the use of stacks to implement recursive ideas in non-recursive languages.

If this program were to be given exciting data like `56 65 9`, then its dynamic execution would result in a stack frame history something like the following, where each line represents the relative layout of the stack frames as the procedures are entered and left.

```
                          Stack grows ---->
start main program        Backwards
call Start                Backwards   Start
call Reverse              Backwards   Start   Reverse
read 56 and recurse       Backwards   Start   Reverse   Start
   and again              Backwards   Start   Reverse   Start   Reverse
read 65 and recurse       Backwards   Start   Reverse   Start   Reverse   Start
   and again              Backwards   Start   Reverse   Start   Reverse   Start   Reverse
read 9, write 9, return   Backwards   Start   Reverse   Start   Reverse   Start
   and again              Backwards   Start   Reverse   Start   Reverse
write 65 and return       Backwards   Start   Reverse   Start
   and again              Backwards   Start   Reverse
write 56 and return       Backwards   Start
   and again              Backwards
```

At *run-time* the actual address of a variable somewhere in memory will have to be found by subtracting an *offset* (which, fortunately, *can* be determined at *compile-time*) from the address of the appropriate stack frame (a value which, naturally but unfortunately, *cannot* be predicted at compile-time). The code generated at compile-time must contain enough information for the run-time system to be able to find (or calculate) the base of the appropriate stack frame when it is needed. This calls for considerable thought.

The run-time stack frames are conveniently maintained as a linked list. As a procedure is called, it can set up (in its frame header) a pointer to the base of the stack frame of the procedure that called

it. This pointer is usually called the **dynamic link**. A pointer to the top of this linked structure - that is, to the base of the most recently activated stack frame - is usually given special status, and is called the **base pointer**. Many modern architectures provide a special machine register especially suited for use in this role; we shall assume that our target machine has such a register (`BP`), and that on procedure entry it will be set to the current value of the stack pointer `SP`, while on procedure exit it will be reset to assume the value of the dynamic link emanating from the frame header.

If a variable is local to the procedure currently being executed, its run-time address will then be given by `BP - Offset`, where `Offset` can be predicted at compile-time. The run-time address of a non-local variable must be obtained by subtracting its `Offset` from an address found by descending a chain of stack frame links. The problem is to know how far to traverse this chain, and at first seems easily solved, since at declaration time we have already made provision to associate a static declaration level with each entry in the symbol table. When faced with the need to generate code to address an identifier, we can surely generate code (at compile-time) which will use this information to determine (at run-time) how far down the chain to go. This distance at first appears to be easily predictable - nothing other than the difference in levels between the level we have reached in compilation, and the level at which the identifier (to which we want to refer) was declared.

This is nearly true, but in fact we cannot simply traverse the dynamic link chain by that number of steps. This chain, as its name suggests, reflects the *dynamic* way in which procedures are *called* and their frames stacked, while the level information in the symbol table is related to the *static* depth of nesting of procedures as they were *declared*. Consider the case when the program above has just read the second data number `65`. At that stage the stack memory would have the appearance depicted in Figure 16.4, where the following should be noted:

- The number (`511`) used as the highest address is simply for illustrative purposes.
- Since we are assuming that the stack pointer `SP` is decremented *before* an item is pushed onto the stack, the base register `BP` will actually point to an address just above the current top stack frame. Similarly, immediately after control has been transferred to a procedure the stack pointer `SP` will actually point to the last local variable.

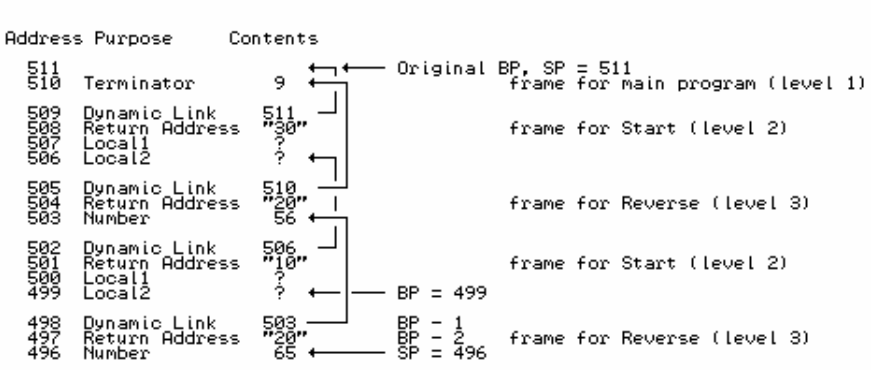


Figure 16.4 Appearance of run time stack after four procedure calls

The compiler would know (at compile-time) that `Terminator` was declared at static level 1, and could have allocated it an offset address of 1 (relative to the base pointer that is used for the main program). Similarly, when parsing the reference to `Terminator` within `Reverse`, the compiler would be aware that it was currently compiling at a static level 3 - a level difference of 2. However, generation of code for descending two steps along the dynamic link chain would result (at run-time) in a memory access to a dynamic link masquerading as a "variable" at location 505, rather than to the variable `Terminator` at location 510.

16.2.2 The static link chain

One way of handling the problem just raised is to provide a second chain for linking data segments, one which will be maintained at run-time using only information that has been embedded in the code generated at compile- time. This second chain is called the **static link chain**, and is set up when a procedure is invoked. By now it should not take much imagination to see that calling a procedure is not handled by simply executing a machine level `JSR` instruction, but rather by the execution of a complex activation and calling sequence.

Procedure activation is that part of the sequence that reserves storage for the frame header and evaluates the actual parameters needed for the call. Parameter handling is to be discussed later, but in anticipation we shall postulate that the calling routine initiates activation by executing code that

- saves the current stack pointer `SP` in a special register known as the **mark stack pointer** `MP`, and then
- decrements `SP` so as to reserve storage for the frame header, before
- dealing with the arrangements for transferring any actual parameters.

When a procedure is called, code is first executed that stores in the first three elements of its activation record

- a *static link* - a pointer to the base of the stack frame for the most recently active instance of the procedure within which its source code was nested;
- a *dynamic link* - a pointer to the base of the stack frame of the calling routine;
- a *return address* - the code address to which control must finally return in the calling routine;

whereafter the `BP` register can be reset to value that was previously saved in `MP`, and control transferred to the main body of the procedure code.

This calling sequence can, in principle, be associated with either the caller or the called routine. Since a routine is defined once, but possibly called from many places, it is usual to associate most of the actions with the called routine. When this code is *generated*, it incorporates (a) the (known) level difference between the static level from which the procedure is to be called and the static level at which it was declared, and (b) the (known) starting address of the executable code. We emphasize that the static link is only set up at run-time, when code is *executed* that follows the extant static chain from the stack frame of the *calling* routine for as many steps as the level difference between the calling and called routine dictates.

Activating and calling procedures is one part of the story. We also need to make provision for accessing variables. To achieve this, the compiler embeds address information into the generated code. This takes the form of pairs of numbers indicating (a) the (known) level difference between the static level from which the variable is being accessed and the static level where it was declared, and (b) the (known) offset displacement that is to be subtracted from the base pointer of the run-time stack frame. When this code is later executed, the level difference information is used to traverse the static link chain when variable address computations are required. In sharp contrast, the dynamic link chain is used, as suggested earlier, only to discard a stack frame at procedure exit.

With this idea, and for the same program as before, the stack would take on the appearance shown in Figure 16.5.

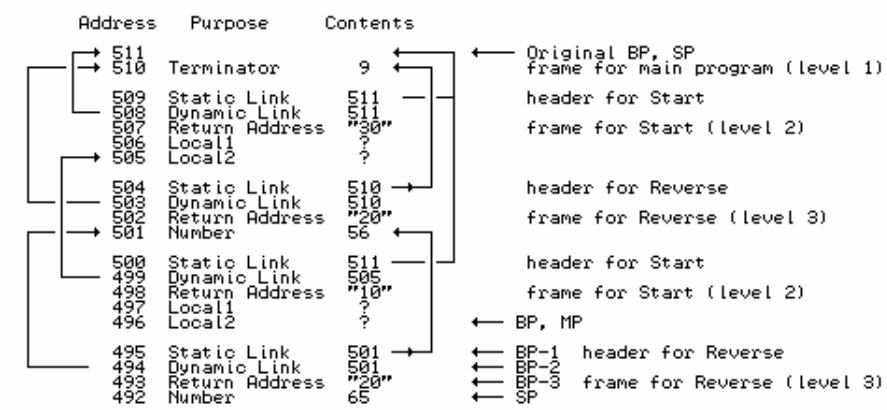


Figure 16.5 Appearance of run time stack after four procedure calls, using static links for non-local variable access

16.2.3 Hypothetical machine support for the static link model

We postulate some extensions to the instruction set of the hypothetical stack machine introduced in section 4.4 so as to support the execution of programs that have simple procedures. We assume the existence of another machine register, the 16-bit `MP`, that points to the frame header at the base of the activation record of the procedure that is about to be called.

One instruction is redefined, and three more are introduced:

`ADR L A`	Push a run-time address onto the stack, for a variable stored at an offset `A` within the stack frame that is `L` steps down the static link chain that begins at the current base register `BP`.
`MST`	Prepare to activate a procedure, saving stack pointer `SP` in `MP`, and then reserving storage for frame header.
`CAL L A`	Call and enter a procedure whose code commences at address `A`, and which was declared at a static difference `L` from the procedure making the call.
`RET`	Return from procedure, resetting `SP, BP` and `PC`.

The extensions to the interpreter of section 4.4 show the detailed operational semantics of these instructions:

```
case STKMC_adr:                         // push run time address
  cpu.sp--;                             // decrement stack pointer
  if (inbounds(cpu.sp))
  { mem[cpu.sp] = base(mem[cpu.pc])     // chain down static links
                + mem[cpu.pc + 1];      // and then add offset
    cpu.pc += 2; }                      // bump program count
  break;
case STKMC_mst:                         // procedure activation
  cpu.mp = cpu.sp;                      // set mark stack pointer
  cpu.sp -= STKMC_headersize;           // bump stack pointer
  inbounds(cpu.sp);                     // check space available
  break;
case STKMC_cal:                         // procedure entry
  mem[cpu.mp - 1] = base(mem[cpu.pc]); // set up static link
  mem[cpu.mp - 2] = cpu.bp;             // save dynamic link
  mem[cpu.mp - 3] = cpu.pc + 2;         // save return address
  cpu.bp = cpu.mp;                      // reset base pointer
  cpu.pc = mem[cpu.pc + 1];             // jump to start of procedure
  break;
case STKMC_ret:                         // procedure exit
  cpu.sp = cpu.bp;                      // discard stack frame
```

```
    cpu.pc = mem[cpu.bp - 3];           // get return address
    cpu.bp = mem[cpu.bp - 2];           // reset base pointer
    break;
```

The routines for calling a procedure and computing the run-time address of a variable make use of the small auxiliary routine `base`:

```
int STKMC::base(int l)
// Returns base of l-th stack frame down the static link chain
{ int current = cpu.bp;                 // start from base pointer
  while (l > 0) { current = mem[current - 1]; l--; }
  return (current);
}
```

16.2.4 Code generation for the static link model

The discussion in the last sections will be made clearer if we examine the refinements to the compiler in more detail.

The routines for parsing the main program and for parsing nested procedures make appropriate entries into the symbol table, and then call upon *Block* to handle the rest of the source code for the routine.

```
Clang
=                           (. TABLE_entries entry; .)
   "PROGRAM"
   Ident<entry.name>        (. entry.idclass = TABLE_progs;
                               Table->enter(entry); Table->openscope(); .)
   WEAK ";"
   Block<entry.level+1, TABLE_progs, 0>
   "." .

ProcDeclaration
=                           (. TABLE_entries entry; .)
   "PROCEDURE"
   Ident<entry.name>        (. entry.idclass = TABLE_procs;
                               CGen->storelabel(entry.p.entrypoint);
                               Table->enter(entry); Table->openscope(); .)
   WEAK ";"
   Block<entry.level+1, entry.idclass, CGEN_headersize>
   ";" .
```

We note that:

- The address of the first instruction in any procedure will be stored in the symbol table in the `entrypoint` field of the entry for the procedure name, and retrieved from there whenever the procedure is to be called.

- The parser for a *Block* is passed a parameter denoting its static level, a parameter denoting its class, and a parameter denoting the offset to be assigned to its first local variable. Offset addresses for variables in the stack frame for a procedure start at 4 (allowing for the size of the frame header), as opposed to 1 (for the main program).

Parsing a *Block* involves several extensions over what was needed when there was only a single main program, and can be understood with reference to the attributed production:

```
Block<int blklevel, TABLE_idclasses blkclass, int initialframesize>
=                           (. int framesize = initialframesize;
                               CGEN_labels entrypoint;
                               CGen->jump(entrypoint, CGen->undefined); .)
   SYNC
   { (   ConstDeclarations
       | VarDeclarations<framesize>
       | ProcDeclaration
     ) SYNC }               (. blockclass = blkclass; blocklevel = blklevel;
                               // global for efficiency
                               CGen->backpatch(entrypoint);
```

```
                              CGen->openstackframe(framesize - initialframesize); .)
   CompoundStatement       (. switch (blockclass)
                              { case TABLE_progs :
                                  CGen->leaveprogram(); break;
                                case TABLE_procs :
                                  CGen->leaveprocedure(); break;
                              }
                              Table->closescope(); .) .
```

in which the following points are worthy of comment:

- Since blocks can be nested, the compiler cannot predict, when a procedure name is *declared*, exactly when the code for that procedure will be *defined*, still less where it will be located in memory. To save a great deal of trouble such as might arise from apparent forward references, we can arrange that the code for each procedure starts with an instruction which may have to branch (over the code for any nested blocks) to the actual code for the procedure body. This initial forward branch is generated by a call to the code generating routine `jump`, and is backpatched when we finally come to generate the code for the procedure body. With a little thought we can see that a simple optimization will allow for the elimination of the forward jump in the common situation where a procedure has no further procedure nested within it. Of course, calls to procedures within which other procedures *are* nested will immediately result in the execution of a further branch instruction, but the loss in efficiency will usually be very small.

- The call to the `openstackframe` routine takes into account the fact that storage will have been allocated for the frame header when a procedure is activated just before it is called.

- The formal parameters `blkclass` and `blklevel` are copied into global variables in the parser to cut down on the number of attributes needed for every other production, and thus improve on parsing efficiency. This rather nasty approach is not needed in Modula-2 and Pascal hand-crafted parsers, where the various routines of the parser can themselves be nested.

- After the *CompoundStatement* has been parsed, code is generated either to halt the program (in the case of a program block), or to effect a procedure return (by calling on `leaveprocedure` to emit a `RET` instruction).

Code for parsing assignments and procedure calls is generated after the LL(1) conflict has been resolved by the call to `Designator`:

```
AssignmentOrCall
=                             (. TABLE_entries entry; .)
   Designator<classset(TABLE_vars, TABLE_procs), entry>
   (   /* assignment */       (. if (entry.idclass != TABLE_vars) SemError(210); .)
       ":=" Expression SYNC   (. CGen->assign(); .)
     | /* procedure call */   (. if (entry.idclass == TABLE_procs)
                                 { CGen->markstack();
                                   CGen->call(blocklevel - entry.level, entry.p.entrypoint);
                                 }
                                 else SemError(210); .)
   ) .
```

This makes use of two routines, `markstack` and `call` that are responsible for generating code for initiating the activation and calling sequences (for our interpretive system these routines simply emit the `MST` and `CAL` instructions). The routine for processing a *Designator* is much as before, save that it must call upon an extended version of the `stackaddress` code generation routine to emit the new form of the `ADR` instruction:

```
Designator<classset allowed, TABLE_entries &entry>
=                             (. TABLE_alfa name;
                                 bool found; .)
   Ident<name>                (. Table->search(name, entry, found);
```

```
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              if (entry.idclass != TABLE_vars) return;
                              CGen->stackaddress(blocklevel - entry.level,
                                                 entry.v.offset); .)
    (   "["                 (. if (entry.v.scalar) SemError(204); .)
        Expression          (. /* determine size for bounds check */
                              CGen->stackconstant(entry.v.size);
                              CGen->subscript(); .)
        "]"
      |                     (. if (!entry.v.scalar) SemError(205); .)
    ) .
```

We observe that an improved code generator could be written to make use of a tree representation for expressions and conditions, similar to the one discussed in section 15.3.2. A detailed Cocol grammar and hand-crafted parsers using this can be found on the source diskette; it suffices to note that virtually no changes have to be made to those parts of the grammar that we have discussed in this section, other than for those responsible for assignment statements.

16.2.5 The use of a "Display"

Another widely used method for handling variable addressing involves the use of a so-called **display**. Since at most one instance of a procedure can be active at one time, only the latest instance of each local variable can be accessible. The tedious business of following a static chain for each variable access at execution time can be eliminated by storing the base pointers for the most recently activated stack frames at each level - the addresses we would otherwise have found after following the static chain - in a small set of dedicated registers. These conceptually form the elements of an array indexed by static level values. Run-time addressing is then performed by subtracting the predicted stack frame offset from the appropriate entry in this array.

When code for a procedure call is required, the interface takes into account the (known) absolute level at which the called procedure was declared, and also the (known) starting address of the executable code. The code to be executed is, however, rather different from that used by the static link method. When a procedure is called it still needs to store the dynamic link, and the return address (in its frame header). In place of setting up the start of the static link chain, the calling sequence updates the display. This turns out to be very easy, as only one element is involved, which can be predicted at compile-time to be the one that corresponds to a static level one higher than that at which the name of the called procedure was declared. (Recall that a *ProcIdentifier* is attributed with the level of the *Block* in which it is declared, and not with the level of the *Block* which defines its local variables and code.)

Similarly, when we leave a procedure, we must not only reset the program counter and base pointer, we may also need to restore a single element of the display. This is strictly only necessary if we have called the procedure from one declared statically at a higher level, but it is simplest to update one element on all returns.

Consequently, when a procedure is called, we arrange for it to store in its frame header:

- a *display copy* - a copy of the current value of the display element for the level one higher than the level of the called routine. This will allow the display to be reset later if necessary.
- a *dynamic link* - a pointer to the base of the stack frame of the calling routine.
- a *return address* - the code address to which control must finally return in the calling routine.

When a procedure relinquishes control, the base pointer is reset from the dynamic link, the program

counter is reset from the return address, and one element of the display is restored from the display copy. The element in question is that for a level one higher than the level at which the name of the called routine was declared, that is, the level at which the block for the routine was compiled, and this level information must be incorporated in the code generated to handle a procedure exit.

Information pertinent to variable addresses is still passed to the code generator by the analyser as pairs of numbers, the first giving the (known) level at which the identifier was declared (an absolute level, not the difference between two levels), and the second giving the (known) offset from the run-time base of the stack frame. This involves only minor changes to the code generation interface so far developed.

This should be clarified by tracing the sequence of procedure calls for the same program as before. When only the main program is active, the situation is as depicted in Figure 16.6(a).

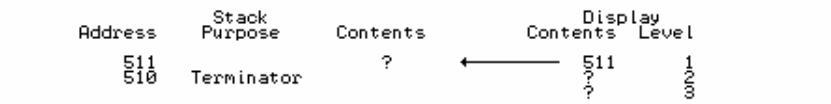


Figure 16.6(a) Stack frames extant immediately after starting execution

After `Start` is activated and called, the situation changes to that depicted in Figure 16.6(b).

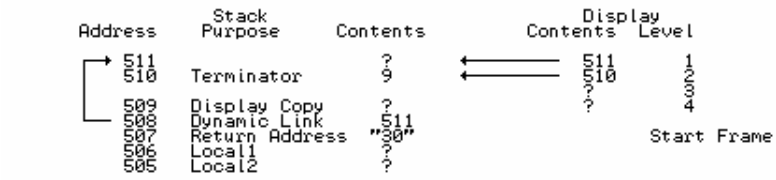


Figure 16.6(b) Stack frames extant immediately after first procedure call

After `Reverse` is called for the first time it changes again to that depicted in Figure 16.6(c).

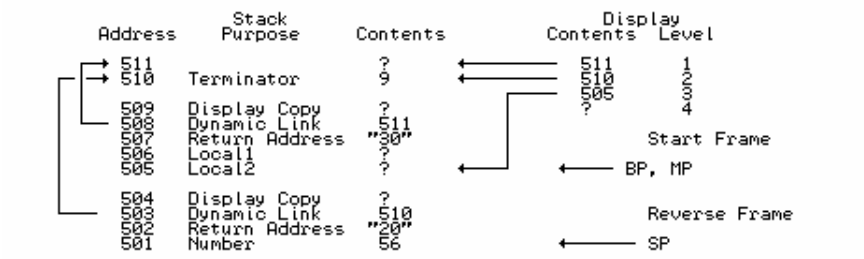


Figure 16.6(c) Stack frames extant immediately after first call to Reverse

After the next (recursive) call to `Start` the changes become rather more significant, as the display copy is now relevant for the first time (Figure 16.6(d)).

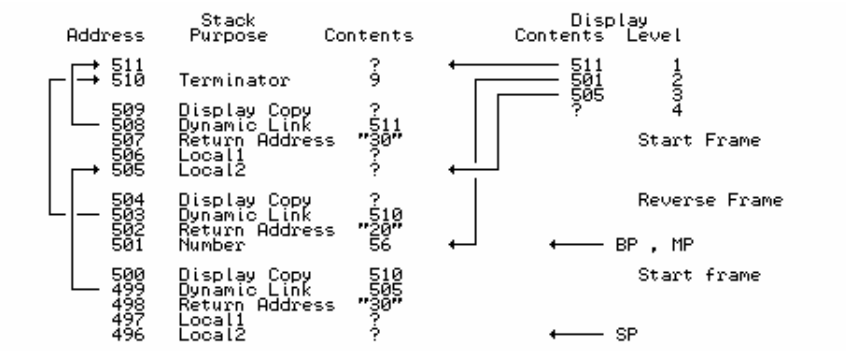


Figure 16.6(d) Stack frames extant immediately after second call to Start

After the next (recursive) call to `Reverse` we get the situation in Figure 16.6(e).

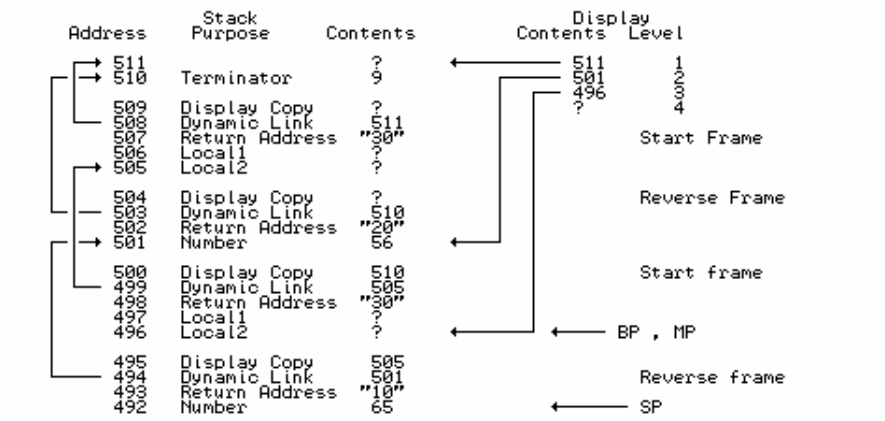


Figure 16.6(e) Stack frames extant immediately after second call to Reverse

When the recursion unwinds, `Reverse` relinquishes control, the stack frame in `495-492` is discarded, and `Display[3]` is reset to `505`. When `Reverse` relinquishes control yet again, the frame in `504-501` is discarded and there is actually no need to alter `Display[3]`, as it is no longer needed. Similarly, after leaving `Start` and discarding the frame in `509-505` there is no need to alter `Display[2]`. However, it is easiest simply to reset the display every time control returns from a procedure.

16.2.6 Hypothetical machine support for the display model

Besides the mark stack register `MP`, our machine is assumed to have a set of display registers, which we can model in an interpreter as a small array, `display`. Conceptually this is indexed from 1, which calls for care in a C++ implementation where arrays are indexed from 0. The `MST` instruction provides support for procedure activation as before, but the `ADR`, `CAL` and `RET` instructions are subtly different:

`ADR`	`L`	`A`	Push a run-time address onto the stack for a variable that was declared at static level `L` and predicted to be stored at an offset `A` from the base of a stack frame.
`CAL`	`L`	`A`	Call and enter a procedure whose *ProcIdentifier* was declared at static level `L`, and whose code commences at address `A`.
`RET`	`L`		Return from a procedure whose *Block* was compiled at level `L`.

The extensions to the interpreter of section 4.4 show the detailed operational semantics of these instructions:

```
 case STKMC_adr:                                // push run time address
   cpu.sp--;                                    // decrement stack pointer
   if (inbounds(cpu.sp))
   { mem[cpu.sp] = display[mem[cpu.pc] - 1]     // extract display element
                 + mem[cpu.pc + 1];             // and then add offset
     cpu.pc += 2; }                             // bump program count
   break;
 case STKMC_cal:                                // procedure entry
   mem[cpu.mp - 1] = display[mem[cpu.pc]];      // save display element
   mem[cpu.mp - 2] = cpu.bp;                    // save dynamic link
   mem[cpu.mp - 3] = cpu.pc + 2;                // save return address
   display[mem[cpu.pc]] = cpu.mp;               // update display
   cpu.bp = cpu.mp;                             // reset base pointer
   cpu.pc = mem[cpu.pc + 1];                    // enter procedure
   break;
 case STKMC_ret:                                // procedure exit
   display[mem[cpu.pc] - 1] = mem[cpu.bp - 1]; // restore display
   cpu.sp = cpu.bp;                             // discard stack frame
   cpu.pc = mem[cpu.bp - 3];                    // get return address
   cpu.bp = mem[cpu.bp - 2];                    // reset base pointer
   break;
```

16.2.7 Code generation for the display model

The attributed productions in a Cocol description of our compiler are very similar to those used in a static link model. The production for *Block* takes into account the new form of the `RET` instruction, and also checks that the limit on the depth of nesting imposed by a finite `display` will not be exceeded:

```
Block<int blklevel, TABLE_idclasses blkclass, int initialframesize>
=                          (. int framesize = initialframesize;
                              CGEN_labels entrypoint;
                              CGen->jump(entrypoint, CGen->undefined);
                              if (blklevel > CGEN_levmax) SemError(213); .)
   SYNC
   { (   ConstDeclarations
       | VarDeclarations<framesize>
       | ProcDeclaration
     ) SYNC }              (. blockclass = blkclass; blocklevel = blklevel;
                              CGen->backpatch(entrypoint);
                              CGen->openstackframe(framesize
                                                   - initialframesize); .)
   CompoundStatement      (. switch (blockclass)
                              { case TABLE_progs :
                                  CGen->leaveprogram(); break;
                                case TABLE_procs :
                                  CGen->leaveprocedure(blocklevel); break;
                              }
                              Table->closescope(); .) .
```

The productions for *AssignmentOrCall* and for *Designator* require trivial alteration to allow for the fact that the code generator is passed absolute static levels, and not level differences:

```
AssignmentOrCall
=                          (. TABLE_entries entry; .)
   Designator<classset(TABLE_vars, TABLE_procs), entry>
   (  /* assignment */      (. if (entry.idclass != TABLE_vars) SemError(210); .)
      ":=" Expression SYNC  (. CGen->assign(); .)
    | /* procedure call */  (. if (entry.idclass == TABLE_procs)
                              { CGen->markstack();
                                CGen->call(entry.level, entry.p.entrypoint);
                              }
                              else SemError(210); .)
   ) .

Designator<classset allowed, TABLE_entries &entry>
=                          (. TABLE_alfa name;
                              bool found; .)
   Ident<name>             (. Table->search(name, entry, found);
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              if (entry.idclass != TABLE_vars) return;
                              CGen->stackaddress(entry.level, entry.v.offset); .)
   (   "["                 (. if (entry.v.scalar) SemError(204); .)
       Expression          (. /* determine size for bounds check */
                              CGen->stackconstant(entry.v.size);
```

```
                              CGen->subscript(); .)
        "]"
      |                     (. if (!entry.v.scalar) SemError(205); .)
    ) .
```

It may be of interest to show the code generated for the program given earlier. The correct Clang source

```
PROGRAM Debug;
  VAR Terminator;

  PROCEDURE Start;
    VAR Local1, Local2;

    PROCEDURE Reverse;
      VAR Number;
      BEGIN
        READ(Number);
        IF Terminator <> Number THEN Start;
        WRITE(Number)
      END;

    BEGIN
      Reverse
    END;

  BEGIN
    Terminator := 9;
    Start
  END.
```

produces the following stack machine code, where for comparison we have shown both models:

```
 Static link       Display

 0 BRN     39        0 BRN     41   jump to start of main program
 2 BRN     32        2 BRN     33   jump to start of Start
 4 DSP      1        4 DSP      1   start of code for Reverse (declared at level 3)
 6 ADR  0  -4        6 ADR  3  -4   address of Number (declared at level 3)
 9 INN                9 INN          read (Number)
10 ADR  2  -1       10 ADR  1  -1   address of Terminator is two levels down
13 VAL              13 VAL          dereference - value of Terminator on stack
14 ADR  0  -4       14 ADR  3  -4   address of Number is on this level
17 VAL              17 VAL          dereference - value of Number now on stack
18 NEQ              18 NEQ          compare for inequality
19 BZE     25       19 BZE     25
21 MST              21 MST          prepare to activate Start
22 CAL  2   2       22 CAL  1   2   recursive call to Start
25 ADR  0  -4       25 ADR  3  -4   address of Number
28 VAL              28 VAL
29 PRN              29 PRN          write(Number)
30 NLN              30 NLN
31 RET              31 RET      3   exit Reverse
32 DSP      2       33 DSP      2   start of code for Start (declared at level 2)
34 MST              35 MST          prepare to activate Reverse
35 CAL  0   4       36 CAL  2   4   call on Reverse, which is declared at this level
38 RET              39 RET      2   exit Start
39 DSP      1       41 DSP      1   start of code for main program (level now 1)
41 ADR  0  -1       43 ADR  1  -1   address of Terminator on stack
44 LIT      9       46 LIT      9   push constant 9 onto stack
46 STO              48 STO          Terminator := 9
47 MST              49 MST          prepare to activate Start
48 CAL  0   2       50 CAL  1   2   call Start, which is declared at this level
51 HLT              53 HLT          stop execution
```

16.2.8 Relative merits of the static link and display models

The display method is potentially more efficient at run-time than the static link method. In some real machines special purpose fast CPU registers may be used to store the display, leading to even greater efficiency. It suffers from the drawback that it seems necessary to place an arbitrary limit on the depth to which procedures may be statically nested. The limit on the size of the display is the same as the maximum static depth of nesting allowed by the compiler at compile-time. Murphy’s Law will ensure that this depth will be inadequate for the program you were going to write to ensure you a niche in the Halls of Fame! Ingenious methods can be found to overcome these

problems, but we leave investigation of these to the exercises that follow.

Exercises

16.9 Since Topsy allows only a non-nested program structure for routines like that found in C and C++, its run-time support system need not be nearly as complex as the one described in this section, although use will still need to be made of the stack frame concept. Discuss the implementation of void functions in Topsy in some detail, paying particular attention to the information that would be needed in the frame header of each routine, and extend your Topsy compiler and the hypothetical machine interpreter to allow you to handle multi-function programs.

16.10 Follow up the suggestion that the display does not have to be restored after every return from a procedure. When should the compiler generate code to handle this operation, and what form should the code take? Are the savings worth worrying about? (The Pascal-S system takes this approach (Wirth, 1981; Rees and Robson, 1987).)

16.11 If you use the display method, is there any real need to use the base register `BP` as well?

16.12 If one studies block-structured programs, one finds that many of the references to variables in a block are either to the local variables of that block, or to the global variables of the main program block. Study the source code for the Modula-2 and Pascal implementation of the hand-crafted parsers and satisfy yourself of the truth of this. If this is indeed so, perhaps special forms of addressing should be used for these variables, so as to avoid the inefficient use of the static link search or display reference at run-time. Explore this idea for the simple compiler-interpreter system we are developing.

16.13 In our emulated machine the computation of every run-time address by invoking a function call to traverse the static link chain might prove to be excessively slow if the idea were extended to a native- code generator. Since references to "intermediate" variables are likely to be less frequent than references to "local" or "global" variables, some compilers (for example, Turbo Pascal) generate code that unrolls the loop implicit in the `base` function for such accesses - that is, they generate an explicit sequence of *N* assignments, rather than a loop that is performed *N* times - thereby sacrificing a marginal amount of space to obtain speed. Explore the implications and implementation of this idea.

16.14 One possible approach to the problem of running out of display elements is to store as large a display as will be needed in the frame header for the procedure itself. Explore the implementation of this idea, and comment on its advantages and disadvantages.

16.15 Are there any dangers lurking behind the peephole optimization suggested earlier for eliminating redundant branch instructions? Consider carefully the code that needs to be generated for an `IF ... THEN ... ELSE` statement.

16.16 Can you think of a way of avoiding the unconditional branch instructions with which nearly every enveloping procedure starts, without using all the machinery of a separate forward reference table?

16.17 Single-pass compilers have difficulty in handling some combinations of mutually recursive procedures. It is not always possible to nest such procedures in such a way that they are always

"declared" before they are "invoked" in the source code - indeed, in C++ it is not possible to nest procedures (functions) at all. The solution usually adopted is to support the *forward* declaration of procedures. In Pascal, and in some Modula-2 compilers this is done by substituting the keyword `FORWARD` for the body of the procedure when it is first declared. In C++ the same effect is achieved through the use of *function prototypes*.

Extend the Clang and Topsy compilers as so far developed so as to allow mutually recursive routines to be declared and elaborated properly. Bear in mind that all procedures declared `FORWARD` *must* later be defined in full, and at the same level as that where the forward declaration was originally made.

16.18 The poor old `GOTO` statement is not only hated by protagonists of structured programming. It is also surprisingly awkward to compile. If you wish to add it to Clang, why should you prevent users from jumping into procedure or function blocks, and if you let them jump out of them, what special action must be taken to maintain the integrity of the stack frame structures?

Further reading

Most texts on compiling block-structured languages give a treatment of the material discussed here, but this will make more sense after the reader has studied the next chapter.

The problems with handling the `GOTO` statement are discussed in the books by Aho, Sethi and Ullman (1986) and Fischer and LeBlanc (1988, 1991).

17 PARAMETERS AND FUNCTIONS

It is the aim of this chapter to show how we can extend our language and its compiler to allow for value-returning functions in addition to regular procedures, and to support the use of parameters. Once again, the syntactic and semantic extensions we shall make are kept as simple as possible, and should be familiar to the reader from a study of other imperative languages.

17.1 Syntax and semantics

The subject of parameter passing is fairly extensive, as the reader may have realized. In the development of programming languages several models of parameter passing have been proposed, and the ones actually implemented vary semantically from language to language, while syntactically often appearing deceptively similar. In most cases, declaration of a subprogram segment is accompanied by the declaration of a list of **formal parameters**, which appear to have a status within the subprogram rather like that of local variables. Invocation of the subprogram is accompanied by a corresponding list of **actual parameters** (sometimes called **arguments**), and it is invariably the case that the relationship between formal and actual parameters is achieved by positional correspondence, rather than by lexical correspondence in the source text. Thus it would be quite legal, if a little confusing to another reader, to declare

```
PROCEDURE AnyName ( A , B )
```

and then to invoke it with a statement of the form

```
AnyName ( B , A )
```

when the `A` in the procedure would be associated with the `B` in the calling routine, and the `B` in the procedure would be associated with the `A` in the calling routine. It may be the lack of name correspondence that is at the root of a great deal of confusion in parameter handling amongst beginners.

The correspondence of formal and actual parameters goes deeper than mere position in a parameter list. Of the various ways in which it might be established, the two most widely used and familiar parameter passing mechanisms are those known as **call-by-reference** and **call-by-value**. In developing the case studies in this text we have, of course, made frequent use of both of methods; we turn now to a discussion of how they are implemented.

The semantics and the implementation of the two mechanisms are quite different:

- In call-by-reference an actual parameter usually takes the form of a *VariableDesignator*. Within the subprogram, a reference to the formal parameter results, at run-time, in a direct reference to the variable designated by the actual parameter, and any change to that formal parameter results in an immediate change to the corresponding actual parameter. In a very real sense, a formal parameter name may be regarded as an *alias* for the actual parameter name. The alias lasts as long as the procedure is active, and may be transmitted to other subprograms with parameters passed in the same way. Call-by-reference is usually accomplished by passing the address associated with the actual parameter to the subprogram for processing.

- In call-by-value, an actual parameter takes the form of an *Expression*. Formal parameters in a subprogram (when declared in this way) are effectively variables local to that subprogram, which start their lives initialized to the values of the corresponding actual parameter expressions. However, any changes made to the values of the formal parameter variables are confined to the subprogram, and cannot be transmitted back via the formal parameters to the calling routine. Fairly obviously, it is the run-time value of the expression which is handed over to the subprogram for processing, rather than an explicit address of a variable.

Call-by-value is preferred for many applications - for example it is useful to be able to pass expressions to procedures like `WRITE` without having to store their values in otherwise redundant variables. However, if an array is passed by value, a complete copy of the array must be passed to the subprogram. This is expensive, both in terms of space and time, and thus many programmers pass all array parameters by reference, even if there is no need for the contents of the array to be modified. In C++, arrays may *only* be passed as reference parameters, although C++ permits the use of the qualifier `const` to prevent an array from being modified in a subprogram. Some languages permit call-by-reference to take place with actual parameters that are expressions in the general sense; in this case the value of the expression is stored in a temporary variable, and the address of that variable is passed to the subprogram.

In what follows we shall partially illustrate both methods, using syntax suggested by C. Simple scalar parameters will be passed by value, and array parameters will be passed by reference in a way that almost models the **open array** mechanism in Modula-2.

We describe the introduction of function and parameter declarations to our language more formally by the following EBNF. The productions are highly non-LL(1), and it should not take much imagination to appreciate that there is now a large amount of context-sensitive information that a practical parser will need to handle (through the usual device of the symbol table). Our productions attempt to depict where such context-sensitivity occurs.

```
ProcDeclaration    =   ( "PROCEDURE" ProcIdentifier | "FUNCTION" FuncIdentifier )
                       [ FormalParameters ] ";"
                       Block  ";" .
FormalParameters   =   "("  OneFormal  { ","  OneFormal }  ")" .
OneFormal          =   ScalarFormal | ArrayFormal .
ScalarFormal       =   ParIdentifier .
ArrayFormal        =   ParIdentifier "[" "]" .
```

We extend the syntax for *ProcedureCall* to allow procedures to be invoked with parameters:

```
ProcedureCall      =   ProcIdentifier ActualParameters .
ActualParameters   =   [ "(" OneActual { "," OneActual } ")" ] .
OneActual          =   ValueParameter | ReferenceParameter .
ValueParameter     =   Expression .
ReferenceParameter =   Variable .
```

We also extend the definition of *Factor* to allow function references to be included in expressions with the appropriate precedence:

```
Factor             =   Variable | ConstIdentifier | number
                     | "(" Expression ")"
                     | FuncIdentifier ActualParameters .
```

and we introduce the *ReturnStatement* in an obvious way:

```
ReturnStatement    =   "RETURN" [ Expression ] .
```

where the *Expression* is only needed within functions, which will be limited (as in traditional C and Pascal) to returning scalar values only. Within a regular procedure the effect of a *ReturnStatement*

is simply to transfer control to the calling routine immediately; within a main program a *ReturnStatement* simply terminates execution.

A simple example of a Clang program that illustrates these extensions is as follows:

```
PROGRAM Debug;

  FUNCTION Last (List[], Limit);
    BEGIN
      RETURN List[Limit];
    END;

  PROCEDURE Analyze (Data[], N);
    VAR
      LocalData[2];
    BEGIN
      WRITE(Last(Data, N+2), Last(LocalData, 1));
    END;

  VAR
    GlobalData[3];

  BEGIN
    Analyze(GlobalData, 1);
  END.
```

The `WRITE` statement in procedure `Analyze` would print out the value of `GlobalData[3]` followed by the value of `LocalData[1]`. `GlobalData` is passed to `Analyze`, which refers to it under the alias of `Data`, and then passes it on to `Last`, which, in turn, refers to it under the alias of `List`.

17.2 Symbol table support for context-sensitive features

It is possible to write a simple context-free set of productions that *do* satisfy the LL(1) constraints, and a Coco/R generated system will require this to be done. We have remarked earlier that it is not possible to specify the requirement that the number of formal and actual parameters must match; this will have to be done by context conditions. So too will the requirement that each actual parameter is passed in a way compatible with the corresponding formal parameter - for example, where a formal parameter is an open array we must not be allowed to pass a scalar variable identifier or an expression as the actual parameter. As usual, a compiler must rely on information stored in the symbol table to check these conditions, and we may indicate the support that must be provided by considering the shell of a simple program:

```
PROGRAM Main;
  VAR G1;                   (* global *)

  PROCEDURE One (P1, P2);   (* two formal scalar parameters *)
    BEGIN                   (* body of One *)
    END;

  PROCEDURE Two;            (* no formal parameters *)
    BEGIN                   (* body of Two *)
    END;

  PROCEDURE Three (P1[]);   (* one formal open array parameter *)
    VAR L1, L2;             (* local to Three *)
    BEGIN                   (* body of Three *)
    END;

  BEGIN                     (* body of Main *)
  END.
```

At the instant where the body of procedure `Three` is being parsed our symbol table might have a structure like that in Figure 17.1.

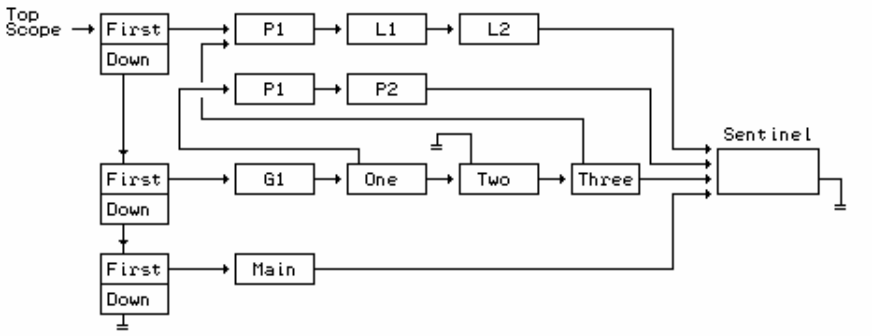


Figure 17.1 A symbol table structure with links from procedure entries to formal parameter entries

Although all three of the procedure identifiers `One, Two` and `Three` are in scope, procedures `One` and `Two` will already have been compiled in a one-pass system. So as to retain information about their formal parameters, internal links are set up from the symbol table nodes for the procedure identifiers to the nodes set up for these parameters. To provide this support it is convenient to extend the definition of the `TABLE_entries` structure:

```
enum TABLE_idclasses
{ TABLE_consts, TABLE_vars, TABLE_progs, TABLE_procs, TABLE_funcs };

struct TABLE_nodes;
typedef TABLE_nodes *TABLE_index;

struct TABLE_entries {
  TABLE_alfa name;             // identifier
  int level;                   // static level
  TABLE_idclasses idclass;     // class
  union {
    struct {
      int value;
    } c;                       // constants
    struct {
      int size, offset;
      bool ref, scalar;
    } v;                       // variables
    struct {
      int params, paramsize;
      TABLE_index firstparam;
      CGEN_labels entrypoint;
    } p;                       // procedures, functions
  };
};
```

Source for an implementation of the `TABLE` class can be found in Appendix B, and it may be helpful to draw attention to the following features:

- Formal parameters are treated within the *Block* of a function or procedure in most cases as though they were variables. So it will be convenient to enter them into the symbol table as such. However, it now becomes necessary to tag the entry for each variable with the extra field `ref`. This denotes whether the identifier denotes a true variable, or is merely an alias for a variable that has been passed to a procedure by reference. Global and local variables and scalar formals will all have this field defined to be `false`.

- Passing an array to a subprogram by *reference* is not simply a matter of passing the address of the first element, even though the subprogram appears to handle open arrays. We shall also need to supply the length of the array (unless we are content to omit array bound checking). This suggests that the value of the `size` field for an array formal parameter can always be 2. We observe that passing open arrays by *value*, as is possible in Modula-2, is likely to be considerably more complicated.

- Formal parameter names, like local variable names, will be entered at a higher level than the procedure or function name, so as to reserve them local status.

- For procedures and functions the `params` field is used to record the number of formal parameters, and the `firstparam` field is used to point to the linked queue of entries for the identifiers that denote the formal parameters. Details of the formal parameters, when needed for context-sensitive checks, can be extracted by further member functions in the `TABLE` class. As it happens, for our simplified system we need only to know whether an actual parameter must be passed by value or by reference, so a simple Boolean function `isrefparam` is all that is required.

- When a subprogram identifier is first encountered, the compiler will not immediately know how many formal parameters will be associated with it. The table handler must make provision for backpatching an entry, and so we need a revised interface to the `enter` routine, as well as an `update` routine:

```
class TABLE {
  public:
    void enter(TABLE_entries &entry, TABLE_index &position);
    // Adds entry to symbol table, and returns its position

    void update(TABLE_entries &entry, TABLE_index position);
    // Updates entry at known position

    bool isrefparam(TABLE_entries &procentry, int n);
    // Returns true if nth parameter for procentry is passed by reference

    // rest as before
};
```

The way in which the declaration of functions and parameters is accomplished may now be understood with reference to the following extract from a Cocol specification:

```
ProcDeclaration
=                          (. TABLE_entries entry; TABLE_index index; .)
   (   "PROCEDURE"         (. entry.idclass = TABLE_procs; .)
     | "FUNCTION"          (. entry.idclass = TABLE_funcs; .)
   ) Ident<entry.name>     (. entry.p.params = 0; entry.p.paramsize = 0;
                              entry.p.firstparam = NULL;
                              CGen->storelabel(entry.p.entrypoint);
                              Table->enter(entry, index);
                              Table->openscope(); .)
   [
   FormalParameters<entry> (. Table->update(entry, index); .)
   ] WEAK ";"
   Block<entry.level+1, entry.idclass, entry.p.paramsize + CGEN_headersize>
   ";" .

FormalParameters<TABLE_entries &proc>
=                          (. TABLE_index p; .)
   "(" OneFormal<proc, proc.p.firstparam>
       { WEAK "," OneFormal<proc, p> } ")" .

OneFormal<TABLE_entries &proc, TABLE_index &index>
=                          (. TABLE_entries formal;
                              formal.idclass = TABLE_vars; formal.v.ref = false;
                              formal.v.size = 1; formal.v.scalar = true;
                              formal.v.offset = proc.p.paramsize
                                              + CGEN_headersize + 1; .)
   Ident<formal.name>
   [ "[" "]"               (. formal.v.size = 2; formal.v.scalar = false;
                              formal.v.ref = true; .)
   ]                       (. Table->enter(formal, index);
                              proc.p.paramsize += formal.v.size;
                              proc.p.params++; .) .
```

Address offsets have to be associated with formal parameters, as with other variables. These are allocated as the parameters are declared. This topic is considered in more detail in the next section; for the moment notice that parameter offsets start at `CGEN_HeaderSize + 1`.

17.3 Actual parameters and stack frames

There are several ways in which actual parameter values may be transmitted to a subprogram. Typically they are pushed onto a stack as part of the activation sequence that is executed before transferring control to the procedure or function which is to use them. Similarly, to allow a function value to be returned, it is convenient to reserve a stack item for this just before the actual parameters are set up, and for the function subprogram to access this reserved location using a suitable offset. The actual parameters might be stored *after* the frame header - that is, within the activation record - or they might be stored *before* the frame header. We shall discuss this latter possibility no further here, but leave the details as an exercise for the curious reader (see Terry (1986) or Brinch Hansen (1985)).

If the actual parameters are to be stored within the activation record, the corresponding formal parameter offsets are easily determined by the procedures specified by the Cocol grammar given earlier. These also keep track of the total space that will be needed for all parameters, and the final offset reached is then passed on to the parser for *Block*, which can continue to assign offsets for local variables beyond this.

To handle function returns it is simplest to have a slightly larger frame header than before. We reserve the first location in a stack frame (that is, at an invariant offset of 1 from the base pointer `BP`) for a function's return value, thereby making code generation for the *ReturnStatement* straightforward. This location is strictly not needed for regular procedures, but it makes for easier code generation to keep all frame headers a constant size. We also need to reserve an element for saving the old mark stack pointer at procedure activation so that it can be restored once a procedure has been completed. We also need to reserve an element for saving the old mark stack pointer at procedure activation so that it can be restored once a procedure has been completed.

If we use the display model, the arrangement of the stack after a procedure has been activated and called will typically be as shown in Figure 17.2. The frame header and actual parameters are set up by the activation sequence, and storage for the local variables is reserved immediately after the procedure obtains control.

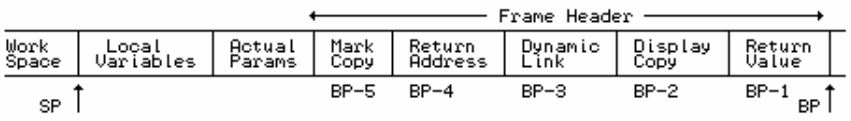


Figure 17.2 Stack frame immediately after a procedure has been called

This may be made clearer by considering some examples. Figure 17.3 shows the layout in memory for the array processing program given in section 17.1, at the instant where function `Last` has just started execution.

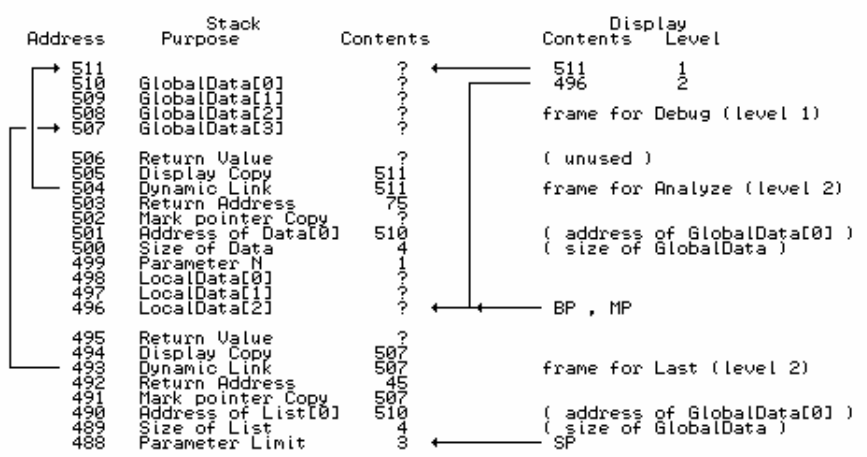

Stack
Address
Purpose
Contents
Display
Contents
Level
511 ? 511 1
510 GlobalData[0] ? 496 2
509 GlobalData[1] ?
508 GlobalData[2] ? frame for Debug (level 1)
507 GlobalData[3] ?
506 Return Value ? (unused)
505 Display Copy 511
504 Dynamic Link 511 frame for Analyze (level 2)
503 Return Address 75
502 Mark pointer Copy ?
501 Address of Data[0] 510 (address of GlobalData[0])
500 Size of Data 4 (size of GlobalData)
499 Parameter N 1
498 LocalData[0] ?
497 LocalData[1] ?
496 LocalData[2] ? BP , MP
495 Return Value ?
494 Display Copy 507
493 Dynamic Link 507 frame for Last (level 2)
492 Return Address 45
491 Mark pointer Copy 507
490 Address of List[0] 510 (address of GlobalData[0])
489 Size of List 4 (size of GlobalData)
488 Parameter Limit 3 SP

Figure 17.3 Arrangement of stack frames after calling a procedure followed by a function

Note that there are three values in the parameter area of the stack frame for `Analyze`. The first two are the actual address of the first element of the array bound to the formal parameter `Data`, and the actual size to be associated with this formal parameter. The third is the initial value assigned to formal parameter `N`. When `Analyze` activates function `Last` it stacks the actual address of the array that was bound to `Data`, as well as the actual size of this array, so as to allow `Last` to bind its formal parameter `List` to the formal parameter `Data`, and hence, ultimately, to the same array (that is, to the global array `GlobalData`).

The second example shows a traditional, if hackneyed, approach to computing factorials:

```
PROGRAM Debug;

  FUNCTION Factorial (M);
    BEGIN
      IF M <= 1 THEN RETURN 1;
      RETURN M * Factorial(M-1);
    END;

  VAR N;

  BEGIN
    READ(N);
    WHILE N > 0 DO
      BEGIN WRITE(Factorial(N)); READ(N) END;
  END.
```

If this program were to be supplied with a data value of `N = 3`, then the arrangement of stack frames would be as depicted in Figure 17.4 immediately after the function has been called for the second time.

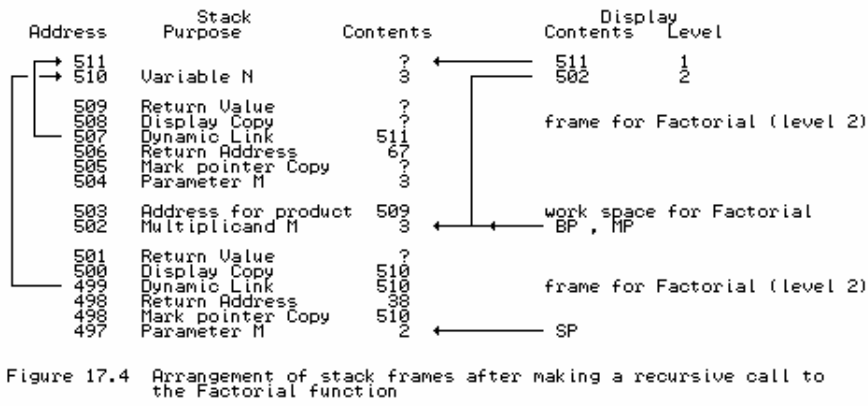

Stack
Address
Purpose
Contents
Display
Contents
Level
511 ? 511 1
510 Variable N 3 502 2
509 Return Value ?
508 Display Copy ? frame for Factorial (level 2)
507 Dynamic Link 511
506 Return Address 67
505 Mark pointer Copy ?
504 Parameter M 3
503 Address for product 509 work space for Factorial
502 Multiplicand M 3 BP , MP
501 Return Value ?
500 Display Copy 510
499 Dynamic Link 510 frame for Factorial (level 2)
498 Return Address 38
498 Mark pointer Copy 510
497 Parameter M 2 SP

Figure 17.4 Arrangement of stack frames after making a recursive call to the Factorial function

`Factorial` can pick up its parameter `M` by using an offset of `5` from `BP`, and can assign the value to be returned to the stack element whose offset is `1` from `BP`. (In practice the addressing might be done via `Display[2]`, rather than via `BP`).

Note that this way of returning function values is entirely consistent with the use of the stack for expression evaluation. In practice, however, many compilers return the value of a scalar function in a machine register.

17.4 Hypothetical stack machine support for parameter passing

Little has to be added to our stack machine to support parameter passing and function handling. Leaving a *Block* is slightly different: after completing a regular procedure we can cut the stack back so as to throw away the entire stack frame, but after completing a function procedure we must leave the return value on the top of stack so that it will be available for incorporation into the expression from which the call was instigated. This means that the `STKMC_ret` instruction requires a second operand. It also turns out to be useful to introduce a `STKMC_nfn` instruction that can be generated at the end of each function block to detect those situations where the flow of control through a function never reaches a *ReturnStatement* (this is very hard to detect at compile-time). Taking into account the increased size of the frame header, the operational semantics of the affected instructions become:

```
case STKMC_cal:                                 // procedure entry
  mem[cpu.mp - 2] = display[mem[cpu.pc]];       // save display element
  mem[cpu.mp - 3] = cpu.bp;                     // save dynamic link
  mem[cpu.mp - 4] = cpu.pc + 2;                 // save return address
  display[mem[cpu.pc]] = cpu.mp;                // update display
  cpu.bp = cpu.mp;                              // reset base pointer
  cpu.pc = mem[cpu.pc + 1];                     // enter procedure
  break;

case STKMC_ret:                                 // procedure exit
  display[mem[cpu.pc] - 1] = mem[cpu.bp - 2]; // restore display
  cpu.mp = mem[cpu.bp - 5];                     // restore mark pointer
  cpu.sp = cpu.bp - mem[cpu.pc + 1];            // discard stack frame
  cpu.pc = mem[cpu.bp - 4];                     // get return address
  cpu.bp = mem[cpu.bp - 3];                     // reset base pointer
  break;

case STKMC_mst:
  if (inbounds(cpu.sp-STKMC_headersize))        // check space available
  { mem[cpu.sp-5] = cpu.mp;                     // save mark pointer
    cpu.mp = cpu.sp;                            // set mark stack pointer
    cpu.sp -= STKMC_headersize;                 // bump stack pointer
  }
  break;

case STKMC_nfn:                                 // bad function (no return)
  ps = badfun; break;                           // change status from running
```

17.5 Context sensitivity and LL(1) conflict resolution

We have already remarked that our language now contains several features that are context-sensitive, and several that make an LL(1) description difficult. These are worth summarizing:

```
    Statement          =   Assignment | ProcedureCall | ...
    Assignment         =   Variable ":="  Expression .
    ProcedureCall      =   ProcIdentifier ActualParameters .
```

Both *Assignment* and *ProcedureCall* start with an *identifier*. Parameters cause similar difficulties:

```
ActualParameters   =   [ "(" OneActual { "," OneActual } ")" ] .
OneActual          =   ValueParameter | ReferenceParameter .
ValueParameter     =   Expression .
ReferenceParameter =   Variable .
```

OneActual is non-LL(1), as *Expression* might start with an *identifier*, and *Variable* certainly does. An *Expression* ultimately contains at least one *Factor*:

```
Factor             =    Variable | ConstIdentifier | number
                      | "(" Expression ")"
                      | FuncIdentifier ActualParameters .
```

and three alternatives in *Factor* start with an identifier. A *Variable* is problematic:

```
Variable           =   VarIdentifier [ "[" Expression "]" ] .
```

In the context of a *ReferenceParameter* the optional index expression is not allowed, but in the context of all other *Factors* it must be present. Finally, even the *ReturnStatement* becomes context-sensitive:

```
ReturnStatement    =   "RETURN" [ Expression ] .
```

In the context of a function *Block* the *Expression* must be present, while in the context of a regular procedure or main program *Block* it must be absent.

17.6 Semantic analysis and code generation

We now turn to a consideration of how the context-sensitive issues can be handled by our parser, and code generated for programs that include parameter passing and value returning functions. It is convenient to consider hand-crafted and automatically generated compilers separately.

17.6.1 Semantic analysis and code generation in a hand-crafted compiler

As it happens, each of the LL(1) conflicts and context-sensitive constraints is easily handled when one writes a hand-crafted parser. Each time an identifier is recognized it is immediately checked against the symbol table, after which the appropriate path to follow becomes clear. We consider the hypothetical stack machine interface once more, and in terms of simplified on-the-fly code generation, making the assumption that the source will be free of syntactic errors. Full source code is, of course, available on the source diskette.

Drawing a distinction between assignments and procedure calls has already been discussed in section 16.1.5, and is handled from within the parser for *Statement*. The parser for *ProcedureCall* is passed the symbol table entry apposite to the procedure being called, and makes use of this in calling on the parser to handle that part of the activation sequence that causes the actual parameters to be stacked before the call is made:

```
void PARSER::ProcedureCall(TABLE_entries entry)
// ProcedureCall = ProcIdentifier ActualParameters .
{ GetSym();
  CGen->markstack();                          // code for activation
  ActualParameters(entry);                    // code to evaluate arguments
  CGen->call(entry.level, entry.p.entrypoint);   // code to transfer control
}
```

A similar extension is needed to the routine that parses a *Factor*:

```
void PARSER::Factor(void)
// Factor = Variable | ConstIdentifier | FuncIdentifier ActualParameters ..
// Variable = Designator .
{ TABLE_entries entry;
  switch (SYM.sym)
  { case SCAN_identifier:                          // several cases arise...
      Table->search(SYM.name, entry);              // look it up
      switch (entry.idclass)                       // resolve LL(1) conflict
      { case TABLE_consts:
          GetSym();
          CGen->stackconstant(entry.c.value);      // code to load named constant
          break;
        case TABLE_funcs:
          GetSym();
          CGen->markstack();                       // code for activation
          ActualParameters(entry);                 // code to evaluate arguments
          CGen->call(entry.level,
                     entry.p.entrypoint);          // code to transfer control
          break;
        case TABLE_vars:
          Designator(entry);                       // code to load address
          CGen->dereference(); break;              // code to load value
      }
      break;
                                                   // ... other cases
  }
}
```

The parsers that handle *ActualParameters* and *OneActual* are straightforward, and make use of the extended features in the symbol table handler to distinguish between reference and value parameters:

```
void PARSER::ActualParameters(TABLE_entries procentry)
// ActualParameters = [ "(" OneActual { "," OneActual } ")" ] .
{ int actual = 0;
  if (SYM.sym == SCAN_lparen)                      // check for any arguments
  { GetSym(); OneActual(procentry, actual);
    while (SYM.sym == SCAN_comma)
    { GetSym(); OneActual(followers, procentry, actual); }
    accept(SCAN_rparen);
  }
  if (actual != procentry.p.params)
    Report->error(209);                            // wrong number of arguments
}

void PARSER::OneActual(TABLE_entries procentry, int &actual)
// OneActual = ArrayIdentifier | Expression .  (depends on context)
{ actual++;                                        // one more argument
  if (Table->isrefparam(procentry, actual))        // check symbol table
    ReferenceParameter();
  else
    Expression();
}
```

The several situations where it is necessary to generate code that will push the run-time address of a variable or parameter onto the stack all depend ultimately on the `stackaddress` routine in the code generator interface. This has to be more complex than before, because in the situations where a variable is really an alias for a parameter that has been passed by reference, the offset recorded in the symbol table is really the offset where one will find yet another address. To push the true address onto the stack requires that we load the address of the offset, and then dereference this to find the address that we really want. Hence the code generation interface takes the form

```
    stackaddress(int level, int offset, bool byref);
```

which, for our stack machine will emit a `LDA level offset` instruction, followed by a `VAL` instruction if `byref` is `true`. This has an immediate effect on the parser for a *Designator*, which now becomes:

```
void PARSER::Designator(TABLE_entries entry)
// Designator = VarIdentifier [ "[" Expression "]" ] .
{ CGen->stackaddress(entry.level, entry.v.offset, entry.v.ref); // base address
  GetSym();
```

```
  if (SYM.sym == SCAN_lbracket)                 // array reference
  { GetSym();
    Expression();                               // code to evaluate index
    if (entry.v.ref)                            // get size from hidden parameter
      CGen->stackaddress(entry.level, entry.v.offset + 1, entry.v.ref);
    else                                        // size known from symbol table
      CGen->stackconstant(entry.v.size);
    CGen->subscript();
    accept(SCAN_rbracket);
  }
}
```

The first call to `stackaddress` is responsible for generating code to push the address of a scalar variable onto the stack, or the address of the first element of an array. If this array has been passed by reference it is necessary to dereference that address to find the true address of the first element of the array, and to determine the true size of the array by retrieving the next (hidden) actual parameter. Another situation in which we wish to push such addresses onto the stack arises when we wish to pass a formal array parameter on to another routine as an actual parameter. In this case we have to push not only the address of the base of the array, but also a second hidden argument that specifies its size. This is handled by the parser that deals with a *ReferenceParameter*:

```
void PARSER::ReferenceParameter(void)
// ReferenceParameter = ArrayIdentifier .  (unsubscripted)
{ TABLE_entries entry;
  Table->search(SYM.name, entry);               // assert : SYM.sym = identifier
  CGen->stackaddress(entry.level, entry.v.offset, entry.v.ref);  // base
                                                // pass size as next parameter
  if (entry.v.ref)                              // get size from formal parameter
    CGen->stackaddress(entry.level, entry.v.offset + 1, entry.v.ref);
  else                                          // size known from symbol table
    CGen->stackconstant(entry.v.size);
  GetSym();                                     // should be comma or rparen
}
```

The variations on the *ReturnStatement* are easily checked, since we have already made provision for each *Block* to be aware of its category. Within a function a *ReturnStatement* is really an assignment statement, with a destination whose address is always at an offset of 1 from the base of the stack frame.

```
void PARSER::ReturnStatement(void)
// ReturnStatement = "RETURN" [ Expression ] .
{ GetSym();                                     // accept RETURN
  switch (blockclass)                           // semantics depend on context
  { case TABLE_funcs:
      CGen->stackaddress(blocklevel, 1, false);  // address of return value
      Expression(followers); CGen->assign();     // code to compute and assign
      CGen->leavefunction(blocklevel); break;    // code to exit function
    case TABLE_procs:
      CGen->leaveprocedure(blocklevel); break;   // direct exit from procedure
    case TABLE_progs:
      CGen->leaveprogram(); break;               // direct halt from main program
  }
}
```

As illustrative examples we give the code for the programs discussed previously:

```
  0 : PROGRAM Debug;
  0 :
  0 :   FUNCTION Factorial (M);
  2 :     BEGIN
  2 :       IF M <= 1 THEN RETURN 1;
 20 :       RETURN M * Factorial(M-1);
 43 :     END;
 44 :
 44 :   VAR N;
 44 :
 44 :   BEGIN
 46 :     READ(N);
 50 :     WHILE N > 0 DO
 59 :       BEGIN WRITE(Factorial(N)); READ(N) END;
 75 :   END.

 0 BRN   44  Jump to start of program       40 RET 2  1    Exit function
```

```
 2 ADR 2 -5  BEGIN Factorial               43 NFN       END Factorial
 5 VAL         Value of M                  44 DSP    1  BEGIN main program
 6 LIT    1                                46 ADR 1 -1     Address of N
 8 LEQ         M <= 1 ?                    49 INN         READ(N)
 9 BZE   20    IF M <= 1 THEN              50 ADR 1 -1     Address of N
11 ADR 2 -1      Address of return val     53 VAL          Value of N
14 LIT    1      Value of 1                54 LIT    0    WHILE N > 0 DO
16 STO           Store as return value     56 GTR
17 RET 2  1      Exit function             57 BZE   75
20 ADR 2 -1    Address of return value     59 MST          Mark stack
23 ADR 2 -5    Address of M                60 ADR 1 -1     Address of N
26 VAL         Value of M                  63 VAL          Value of N (argument)
27 MST         Mark stack                  64 CAL 1  2     Call Factorial
28 ADR 2 -5    Address of M                67 PRN          WRITE(result)
31 VAL         Value of M                  68 NLN
32 LIT    1                                69 ADR 1 -1
34 SUB         Value of M-1 (argument)     72 INN          READ(N)
35 CAL 1  2    Recursive call              73 BRN   50    END
38 MUL         Value M*Factorial(M-1)      75 HLT       END
39 STO         Store as return value
```

```
 0 : PROGRAM Debug;
 2 :
 2 :   FUNCTION Last (List[], Limit);
 2 :     BEGIN
 2 :       RETURN List[Limit];
23 :     END;
24 :
24 :   PROCEDURE Analyze (Data[], N);
24 :     VAR
26 :       LocalData[2];
26 :     BEGIN
26 :       Write(Last(Data, N+2), Last(LocalData, 1));
59 :     END;
62 :
62 :   VAR
62 :     GlobalData[3];
62 :
62 :   BEGIN
64 :     Analyze(GlobalData, 1);
75 :   END.
```

```
 0 BRN   62  Jump to start of program      38 VAL         Value of N
 2 ADR 2 -1      Address of return value   39 LIT    2
 5 ADR 2 -5                                41 ADD         Value of N+2 (argument)
 8 VAL           Address of List[0]        42 CAL 1  2    Last(Data, N+2)
 9 ADR 2 -7      Address of Limit          45 PRN         Write result
12 VAL           Value of Limit            46 MST         Mark Stack
13 ADR 2 -6                                47 ADR 2 -8    Address of LocalData[0]
16 VAL           Size of List              50 LIT    3    Size of LocalData
17 IND           Subscript                 52 LIT    1    Value 1 (parameter)
18 VAL           Value of List[Limit]      54 CAL 1  2    Last(LocalData, 1)
19 STO         Store as return value       57 PRN         Write result
20 RET 2  1    and exit function           58 NLN         WriteLn
23 NFN       END Last                      59 RET 2  0  END Analyze
24 DSP    3  BEGIN Analyze                 62 DSP    4  BEGIN Debug
26 MST           Mark Stack                64 MST         Mark stack
27 ADR 2 -5      First argument is         65 ADR 1 -1    Address of GlobalData[0]
30 VAL           Address of Data[0]        68 LIT    4    Size of GlobalData
31 ADR 2 -6      Hidden argument is        70 LIT    1    Value 1 (argument)
34 VAL           Size of Data              72 CAL 1 24    Analyze(GlobalData, 1)
35 ADR 2 -7      Compute last argument     75 HLT       END
```

17.6.2 Semantic analysis and code generation in a Coco/R generated compiler

If we wish to write an LL(1) grammar as input for Coco/R, things become somewhat more complex. We are obliged to write our productions as

```
Statement          =   AssignmentOrCall | ...
AssignmentOrCall   =   Designator ( ":=" Expression | ActualParameters ) .
ActualParameters   =   [ "(" OneActual { "," OneActual } ")" ] .
OneActual          =   Expression .
Factor             =      Designator ActualParameters | number
                       | "(" Expression ")" .
Designator         =   identifier [ "[" Expression "]" ] .
ReturnStatement    =   "RETURN" [ Expression ] .
```

This implies that *Designator* and *Expression* have to be attributed rather cleverly to allow all the

conflicts to be resolved. This can be done in several ways. We have chosen to illustrate a method where the routines responsible for parsing these productions are passed a Boolean parameter stipulating whether they are being called in a context that requires that the appearance of an array name must be followed by a subscript (this is always the case except where an actual parameter is syntactically an expression, but must semantically be an unsubscripted array name). On its own this system is still inadequate for constraint analysis, and we must also provide some method for checking whether an expression used as an actual reference parameter is comprised only of an unsubscripted array name.

At the same time we may take the opportunity to discuss the use of an AST as an intermediate representation of the semantic structure of a program, by extending the treatment found in section 15.3.2. The various node classes introduced in that section are extended and enhanced to support the idea of a node to represent a procedure or function call, linked to a set of nodes each of which represents an actual parameter, and each of which, in turn, is linked to the tree structure that represents the expression associated with that actual parameter. The sort of structures we set up are exemplified in Figure 17.5, which depicts an AST corresponding to the procedure call in the program outlined below

```
PROGRAM Debug;

  FUNCTION F (X);
    BEGIN END;              (* body of F *)

  PROCEDURE P (U, V[], W);
    BEGIN END;              (* body of P *)

  VAR
    X, Y, A[7];
  BEGIN
    P(F(X+5), A, Y)
  END.
```

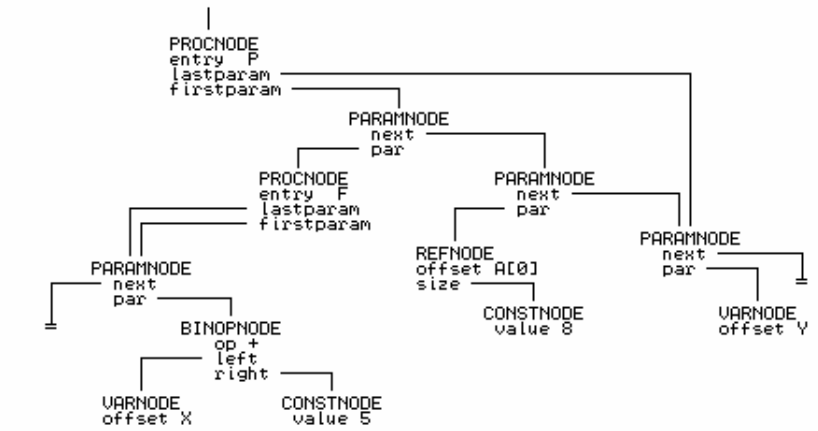


Figure 17.5 AST structures for the statement P(F(X+5), A, Y)

Our base `NODE` class is extended slightly from the one introduced earlier, and now incorporates a member for linking nodes together when they are elements of argument lists:

```
struct NODE {
  int value;                    // value to be associated with this node
  bool defined;                 // true if value predictable at compile time
  bool refnode;                 // true if node corresponds to a ref parameter
  NODE()                        { defined = false; refnode = false; }
  virtual void emit1(void)    = 0;
  virtual void emit2(void)    = 0;
  virtual void link(AST next) = 0;
};
```

Similarly, the VARNODE class has members to record the static level, and whether the corresponding variable is a variable in its own right, or is simply an alias for an array passed by reference:

```
struct VARNODE : public NODE {
  bool ref;                     // direct or indirectly accessed
  int level;                    // static level of declaration
  int offset;                   // offset of variable assigned by compiler
  VARNODE() {;}                 // default constructor
  VARNODE(bool R, int L, int O) { ref = R; level = L; offset = O; }
  virtual void emit1(void);     // generate code to retrieve value of variable
  virtual void emit2(void);     // generate code to retrieve address of variable
  virtual void link(AST next)   {;}
};
```

Procedure and function calls give rise to instances of a PROCNODE class. Such nodes need to record the static level and entry point of the routine, and have further links to the nodes that are set up to represent the queue of actual parameters or arguments. It is convenient to introduce two such pointers so as to simplify the link member function that is responsible for building this queue.

```
struct PROCNODE : public NODE {
  int level, entrypoint;        // static level, address of first instruction
  AST firstparam, lastparam;    // pointers to argument list
  PROCNODE(int L, int E)
    { level = L; entrypoint = E; firstparam = NULL; lastparam = NULL; }
  virtual void emit1(void);     // generate code for procedure/function call
  virtual void emit2(void)      {;}
  virtual void link(AST next);  // link next actual parameter
};
```

The actual arguments give rise to nodes of a new PARAMNODE class. As can be seen from Figure 17.5, these require pointer members: one to allow the argument to be linked to another argument, and one to point to the expression tree for the argument itself:

```
struct PARAMNODE : public NODE {
  AST par, next;                // pointers to argument and to next argument
  PARAMNODE(AST P)              { par = P; next = NULL; }
  virtual void emit1(void);     // push actual parameter onto stack
  virtual void emit2(void)      {;}
  virtual void link(AST param)  { next = param; }
};
```

Actual parameters are syntactically expressions, but we need a further REFNODE class to handle the passing of arrays as actual parameters:

```
struct REFNODE : public VARNODE {
  AST size;                     // real size of array argument
  REFNODE(bool R, int L, int O, AST S)
    { ref = R; level = L; offset = O; size = S; refnode = true; }
  virtual void emit1(void);     // generate code to push array address, size
  virtual void emit2(void)      {;}
  virtual void link(AST next)   {;}
};
```

Tree building operations may be understood by referring to the attributes with which a Cocol specification would be decorated:

```
AssignmentOrCall
=                           (. TABLE_entries entry; AST des, exp;.)
   Designator<des, classset(TABLE_vars, TABLE_procs), entry, true>
   (   /* assignment */     (. if (entry.idclass != TABLE_vars) SemError(210); .)
       ":=" Expression<exp, true>
       SYNC                 (. CGen->assign(des, exp); .)
     | /* procedure call */ (. if (entry.idclass < TABLE_procs)
                                 { SemError(210); return; }
                               CGen->markstack(des, entry.level,
                                               entry.p.entrypoint); .)
       ActualParameters<des, entry>
                            (. CGen->call(des); .)
   ) .


Designator<AST &D, classset allowed, TABLE_entries &entry, bool entire>
```

```
=                          (. TABLE_alfa name; AST index, size;
                              bool found;
                              D = CGen->emptyast(); .)
   Ident<name>             (. Table->search(name, entry, found);
                              if (!found) SemError(202);
                              if (!allowed.memb(entry.idclass)) SemError(206);
                              if (entry.idclass != TABLE_vars) return;
                              CGen->stackaddress(D, entry.level,
                                                 entry.v.offset, entry.v.ref); .)
   (   "["                 (. if (entry.v.scalar) SemError(204); .)
       Expression<index, true>
                           (. if (!entry.v.scalar)
                              /* determine size for bounds check */
                              { if (entry.v.ref)
                                  CGen->stackaddress(size, entry.level,
                                                     entry.v.offset + 1, false);
                                else
                                  CGen->stackconstant(size, entry.v.size);
                                CGen->subscript(D, entry.v.ref, entry.level,
                                                entry.v.offset, size, index);
                              } .)
       "]"
     |                     (. if (!entry.v.scalar)
                              { if (entire) SemError(205);
                                if (entry.v.ref)
                                  CGen->stackaddress(size, entry.level,
                                                     entry.v.offset + 1, false);
                                else
                                  CGen->stackconstant(size, entry.v.size);
                                CGen->stackreference(D, entry.v.ref, entry.level,
                                                     entry.v.offset, size);
                              } .)
   ) .


ActualParameters<AST &p, TABLE_entries proc>
=                          (. int actual = 0; .)
   [  "("                  (. actual++; .)
      OneActual<p, (*Table).isrefparam(proc, actual)>
      { WEAK "," 	   (. actual++; .)
      OneActual<p, (*Table).isrefparam(proc, actual)> } ")"
   ]                       (. if (actual != proc.p.params) SemError(209); .) .


OneActual<AST &p, bool byref>
=                          (. AST par; .)
   Expression<par, !byref> (. if (byref && !CGen->isrefast(par)) SemError(214);
                              CGen->linkparameter(p, par); .) .

ReturnStatement
=                          (. AST dest, exp; .)
   "RETURN"
   (                       (. if (blockclass != TABLE_funcs) SemError(219);
                              CGen->stackaddress(dest, blocklevel, 1, false); .)
       Expression<exp, true>
                           (. CGen->assign(dest, exp);
                              CGen->leavefunction(blocklevel); .)
     | /* empty */         (. switch (blockclass)
                              { case TABLE_procs :
                                  CGen->leaveprocedure(blocklevel); break;
                                case TABLE_progs :
                                  CGen->leaveprogram(); break;
                                case TABLE_funcs :
                                  SemError(220); break;
                              } .)
   ) .

Expression<AST &E, bool entire>
=                          (. AST T; CGEN_operators op;
                              E = CGen->emptyast(); .)
   (   "+" Term<E, true>
     | "-" Term<E, true>   (. CGen->negateinteger(E); .)
     | Term<E, entire>
   )
   { AddOp<op> Term<T, true>(. CGen->binaryintegerop(op, E, T); .)
   } .

Term<AST &T, bool entire>
=                          (. AST F; CGEN_operators op; .)
   Factor<T, entire>
   { (  MulOp<op>
      | /* missing op */   (. SynError(92); op = CGEN_opmul; .)
     ) Factor<F, true>     (. CGen->binaryintegerop(op, T, F); .)
```

```
   } .

Factor<AST &F, bool entire>
=                            (. TABLE_entries entry;
                                int value;
                                F = CGen->emptyast(); .)
     Designator<F, classset(TABLE_consts, TABLE_vars, TABLE_funcs), entry, entire>
                             (. switch (entry.idclass)
                                { case TABLE_consts :
                                    CGen->stackconstant(F, entry.c.value); return;
                                  case TABLE_procs :
                                  case TABLE_funcs :
                                    CGen->markstack(F, entry.level,
                                                    entry.p.entrypoint); break;
                                  case TABLE_vars :
                                  case TABLE_progs :
                                    return;
                                 } .)
     ActualParameters<F, entry>
   | Number<value>           (. CGen->stackconstant(F, value); .)
   | "(" Expression<F, true> ")" .
```

The reader should compare this with the simpler attributed grammar presented in section 15.3.2, and take note of the following points:

- All productions that have to deal with identifiers call upon *Designator*. So far as code generation is concerned, this production is responsible for creating nodes that represent the addresses of variables. Where other identifiers are recognized, execution of a `return` bypasses code generation, and leaves the routine after retrieving the symbol table entry for that identifier.

- *Designator* must now permit the appearance of an unsubscripted array name, creating an instance of a `REFNODE` in this case. Note the use of the `entire` parameter passed to *Designator*, *Expression*, *Term* and *Factor* to enable checking of the context in which the subscript may be omitted.

- Parsing of *OneActual* is simply effected by a call to *Expression*. After this parsing is completed, a check must be carried out to see whether a reference parameter does, in fact, consist only of an unsubscripted array name. Notice that *OneActual* also incorporates a call to a new code generating routine that will link the node just created for the actual parameter to the parameter list emanating from the node for the procedure itself, a node that was created by the `markstack` routine.

- Productions like *AssignmentOrCall* and *Factor* follow the call to *Designator* with tests on the class of the identifier that has been recognized, and use this information to drive the parse further (in *Factor*) or to check constraints (in *AssignmentOrCall*).

As before, once a AST structure has been built, it can be traversed and the corresponding code generated by virtue of each node "knowing" how to generate its own code. It will suffice to demonstrate two examples. To generate code for a procedure call for our hypothetical stack machine we define the `emit1` member function to be

```
void PROCNODE::emit1(void)
// generate procedure/function activation and call
{ CGen->emit(int(STKMC_mst));
  if (firstparam) { firstparam->emit1(); delete firstparam; }
  CGen->emit(int(STKMC_cal));
  CGen->emit(level);
  CGen->emit(entrypoint);
}
```

which, naturally, calls on the `emit1` member of its first parameter to initiate the stacking of the actual parameters as part of the activation sequence. This member, in turn, calls on the `emit1`

member of its successor to handle subsequent arguments:

```
void PARAMNODE::emit1(void)
// push actual parameter onto stack during activation
{ if (par) { par->emit1(); delete par; }     // push this argument
  if (next) { next->emit1(); delete next; }  // follow link to next argument
}
```

Source code for the complete implementation of the code generator class can be found in Appendix C and also on the source diskette, along with implementations for hand-crafted compilers that make use of tree structures, and implementations that make use of the traditional variant records or unions to handle the inhomogeneity of the tree nodes.

Exercises

17.1 Some authors suggest that value-returning function subprograms are not really necessary; one can simply use procedures with call-by-reference parameter passing instead. On the other hand, in C++ all subprograms are potentially functions. Examine the relative merits of providing both in a language, from the compiler writer's and the user's viewpoints.

17.2 Extend Topsy and its compiler to allow functions and procedures to have parameters. Can you do this in such a way a function can be called either as an operand in an expression, or as a stand-alone statement, as in C++?

17.3 The usual explanation of call-by-value leaves one with the impression that this mode of passing is very safe, in that changes within a subprogram can be confined to that subprogram. However, if the value of a pointer variable is passed by value this is not quite the whole story. C does not provide call-by- reference, because the same effect can be obtained by writing code like

```
void swap (int *x, int *y)
{ int z; z = *x; *x = *y; *y = z; }
```

Extend Topsy to provide explicit operators for computing an address, and dereferencing an address (as exemplified by `&variable` and `*variable` in C), and use these features to provide a reference passing mechanism for scalar variables. Is it possible to make these operations secure (that is, so that they cannot be abused)? Are any difficulties caused by overloading the asterisk to mean multiplication in one context and dereferencing an address in another context?

17.4 The array passing mechanisms we have devised effectively provide the equivalent of Modula-2's "open" array mechanism for arrays passed by reference. Extend Clang and its implementation to provide the equivalent of the `HIGH` function to complete the analogy.

17.5 Implement parameter passing in Clang in another way - use the Pascal/Modula convention of preceding formal parameters by the keyword `VAR` if the call-by-reference mechanism is to be used. Pay particular attention to the problems of array parameters.

17.6 In Modula-2 and Pascal, the keyword `VAR` is used to denote call-by-reference, but no keyword is used for the (default) call-by-value. Why does this come in for criticism? Is the word `VAR` a good choice?

17.7 How do you cater for forward declaration of functions and procedures when you have to take formal parameters into account (see Exercise 16.17)?

17.8 (Longer) If you extend Clang or Topsy to introduce a Boolean type as well as an integer one (see Exercise 14.30), how do you solve the host of interesting problems that arise when you wish to introduce Boolean functions and Boolean parameters?

17.9 Follow up the suggestion that parameters can be evaluated before the frame header is allocated, and are then accessed through positive offsets from the base register `BP`.

17.10 Exercise 15.16 suggested the possibility of peephole optimization for replacing the common code sequence for loading an address and then dereferencing this, assuming the existence of a more powerful `STKMC_psh` operation. How would this be implemented when procedures, functions, arrays and parameters are involved?

17.11 In previous exercises we have suggested that undeclared identifiers could be entered into the symbol table at the point of first declaration, so as to help with suppressing further spurious errors. What is the best way of doing this if we might have undeclared variables, arrays, functions, or procedures?

17.12 (Harder) Many languages allow formal parameters to be of a procedure type, so that procedures or functions may be passed as actual parameters to other routines. C++ allows the same effect to be achieved by declaring formal parameters as pointers to functions. Can you extend Clang or Topsy to support this feature? Be careful, for the problem might be more difficult than it looks, except for some special simple cases.

17.13 Introduce a few standard functions and procedures into your languages, such as the `ABS`, `ODD` and `CHR` of Modula-2. Although it is easier to define these names to be reserved keywords, introduce them as pervasive (predeclared) identifiers, thus allowing them to be redeclared at the user's whim.

17.14 It might be thought that the constraint analysis on actual parameters in the Cocol grammar could be simplified so as to depend only on the `entire` parameter passed to the various parsing routines, without the need for a check to be carried out after an *Expression* had been parsed. Why is this check needed?

17.15 If you study the interpreter that we have been developing, you should be struck by the fact that this does a great deal of checking that the stack pointer stays within bounds. This check is strictly necessary, although unlikely to fail if the memory is large enough. It would probably suffice to check only for opcodes that push a value or address onto the stack. Even this would severely degrade the efficiency of the interpreter. Suggest how the compiler and run-time system could be modified so that at compile-time a prediction is made of the extra depth needed by the run-time stack by each procedure. This will enable the run-time system to do a single check that this limit will not be exceeded, as the procedure or program begins execution. (A system on these lines is suggested by Brinch Hansen (1985)).

17.16 Explore the possibility of providing a fairly sophisticated post-mortem dump in the extended interpreter. For example, provide a trace of the subprogram calls up to the point where an error was detected, and give the values of the local variables in each stack frame. To be really user-friendly the run-time system will need to refer to the user names for such entities. How would this alter the whole implementation of the symbol table?

17.17 Now that you have a better understanding of how recursion is implemented, study the compiler you are writing with new interest. It uses recursion a great deal. How deeply do you

suppose this recursion goes when the compiler executes? Is recursive descent "efficient" for all aspects of the compiling process? Do you suppose a compiler would ever run out of space in which to allocate new stack frames for itself when it was compiling large programs?

Further reading

As already mentioned, most texts on recursive descent compilers for block-structured languages treat the material of the last few sections in fair detail, discussing one or other approach to stack frame allocation and management. You might like to consult the texts by Fischer and LeBlanc (1988, 1991), Watson (1989), Elder (1994) or Wirth (1996). The special problem of procedural parameters is discussed in the texts by Aho, Sethi and Ullman (1986) and Fischer and LeBlanc (1988, 1991). Gough and Mohay (1988) discuss the related problem of procedure variables as found in Modula-2.

17.7 Language design issues

In this section we wish to explore a few of the many language design issues that arise when one introduces the procedure and function concepts.

17.7.1 Scope rules

Although the scope rules we have discussed probably seem sensible enough, it may be of interest to record that the scope rules in Pascal originally came in for extensive criticism, as they were incompletely formulated, and led to misconceptions and misinterpretation, especially when handled by one-pass systems. Most of the examples cited in the literature have to do with the problems associated with types, but we can give an example more in keeping with our own language to illustrate a typical difficulty. Suppose a compiler were to be presented with the following:

```
PROGRAM One;

  PROCEDURE Two (* first declared here *);
    BEGIN
      WRITE('First Two')
    END (* Two *);

  PROCEDURE Three;

    PROCEDURE Four;
      BEGIN
        Two
      END (* Four *);

    PROCEDURE Two (* then redeclared here *);
      BEGIN
        WRITE('Second Two')
      END (* Two *);

    BEGIN
      Four; Two
    END (* Three *);

  BEGIN
    Three
  END (* One *).
```

At the instant where procedure `Four` is being parsed, and where the call to `Two` is encountered, the first procedure `Two` (in the symbol table at level 1) seems to be in scope, and code will presumably

be generated for a call to this. However, perhaps the second procedure `Two` should be the one that is in scope for procedure `Four`; one interpretation of the scope rules would require code to be generated for a call to this. In a one-pass system this would be a little tricky, as this second procedure `Two` would not yet have been encountered by the compiler - but note that it would have been by the time the calls to `Four` and `Two` were made from procedure `Three`.

This problem can be resolved to the satisfaction of a compiler writer if the scope rules are formulated so that the scope of an identifier extends from the point of its declaration to the end of the block in which it is declared, and not over the whole block in which it is declared. This makes for easy one-pass compilation, but it is doubtful whether this solution would please a programmer who writes code such as the above, and falls foul of the rules without the compiler reporting the fact.

An ingenious way for a single-pass compiler to check that the scope of an identifier extends over the whole of the block in which it has been declared was suggested by Sale (1979). The basic algorithm requires that every block be numbered sequentially as it compiled (notice that these numbers do not represent nesting levels). Each identifier node inserted into the symbol table has an extra numeric attribute. This is originally defined to be the unique number of the block making the insertion, but each time that the identifier is *referenced* thereafter, this attribute is reset to the number of the block making the reference. Each time an identifier is *declared*, and needs to be entered into the table, a search is made of all the identifiers that are in scope to see if a duplicate identifier entry can be found that is already attributed with a number equal to or greater than that of the block making the declaration. If this search succeeds, it implies that the scope rules are about to be violated. This simple scheme has to be modified, of course, if the language allows for legitimate forward declarations and function prototypes.

17.7.2 Function return mechanisms

Although the use of an explicit *ReturnStatement* will seem natural to a programmer familiar with Modula-2 or C++, it is not the only device that has been explored by language designers. In Pascal, for example, the value to be returned must be defined by means of what appears to be an assignment to a variable that has the same name as the function. Taken in conjunction with the fact that in Pascal a parameterless function call also looks like a variable access, this presents numerous small difficulties to a compiler writer, as a study of the following example will reveal

```
PROGRAM Debug;
  VAR B, C;

  FUNCTION One (W);
    VAR X, Y;

    FUNCTION Two (Z);

      FUNCTION Three;
        BEGIN
          Two := B + X;    (* should this be allowed ? *)
          Three := Three;  (* syntactically correct, although useless *)
        END;

      BEGIN
        Two := B + Two(4); (* must be allowed *)
        Two := B + X;      (* must be allowed *)
        Two := Three;      (* must be allowed *)
        Three := 4;        (* Three is in scope, but cannot be used like this *)
      END;

    BEGIN
      Two := B + X;        (* Two is in scope, but cannot be used like this *)
      X := Two(Y);         (* must be allowed *)
    END;
```

```
BEGIN
  One(B)
END.
```

Small wonder that in his later language designs Wirth adopted the explicit `return` statement. Of course, even this does not find favour with some structured language purists, who preach that each routine should have exactly one entry point and exactly one exit point.

Exercises

17.18 Submit a program similar to the example in section 17.7.1 to any compilers you may be using, and detect which interpretation they place on the code.

17.19 Implement the Sale algorithm in your extended Clang compiler. Can the same sort of scope conflicts arise in C++, and if so, can you find a way to ensure that the scope of an identifier extends over the whole of the block in which it is declared, rather than just from the point of declaration onwards?

17.20 The following program highlights some further problems with interpreting the scope rules of languages when function return values are defined by assignment statements.

```
PROGRAM Silly;

  FUNCTION F;

    FUNCTION F (F) (* nested, and same parameter name as function *);
      BEGIN
        F := 1
      END (* inner F *);

    BEGIN (* outer F *)
      F := 2
    END (* outer F *);

  BEGIN
    WRITE(F)
  END (* Silly *).
```

What would cause problems in one-pass (or any) compilation, and what could a compiler writer do about solving these?

17.21 Notwithstanding our comments on the difficulties of using an assignment statement to specify the value to be returned from a function, develop a version of the Clang compiler that incorporates this idea.

17.22 In Modula-2, a procedure declaration requires the name of the procedure to be quoted again after the terminating `END`. Of what practical benefit is this?

17.23 In classic Pascal the ordering of the components in a program or procedure block is very restrictive. It may be summarized in EBNF on the lines of

```
Block  =  [ ConstDeclarations ]
          [ TypeDeclarations ]
          [ VarDeclarations ]
          { ProcDeclaration }
          CompoundStatement .
```

In Modula-2, however, this ordering is highly permissive:

```
Block  =  { ConstDeclarations | TypeDeclarations | VarDeclarations | ProcDeclaration }
```

```
          CompoundStatement .
```

Oberon (Wirth, 1988b) introduced an interesting restriction:

```
Block  =  { ConstDeclarations | TypeDeclarations | VarDeclarations }
          { ProcDeclaration }
          CompoundStatement .
```

Umbriel (Terry, 1995) imposes a different restriction:

```
Block  =  { ConstDeclarations | TypeDeclarations | ProcDeclaration }
          { VarDeclarations }
          CompoundStatement .
```

Although allowing declarations to appear in any order makes for the simplest grammar, languages that insist on a specific order presumably do so for good reasons. Can you think what these might be?

17.24 How would you write a Cocol grammar or a hand-crafted parser to insist on a particular declaration order, and yet recover satisfactorily if declarations were presented in any order?

17.25 Originally, in Pascal a function could only return a scalar value, and not, for example, an `ARRAY, RECORD` or `SET`. Why do you suppose this annoying restriction was introduced? Is there any easy (legal) way around the problem?

17.26 Several language designers decry function subprograms for the reason that most languages do not prevent a programmer from writing functions that have *side-effects*. The program below illustrates several esoteric side-effects. Given that one really wishes to prevent these, to what extent can a compiler detect them?

```
PROGRAM Debug;
  VAR
    A, B[12];

  PROCEDURE P1 (X[]);
    BEGIN
      X[3] := 1 (* X is passed by reference *)
    END;

  PROCEDURE P2;
    BEGIN
      A := 1 (* modifies global variable *)
    END;

  PROCEDURE P3;
    BEGIN
      P2 (* indirect attack on a global variable *)
    END;

  PROCEDURE P4;
    VAR C;

    FUNCTION F (Y[]);
      BEGIN
        A := 3     (* side-effect *);
        C := 4     (* side-effect *);
        READ(A)    (* side-effect *);
        Y[4] := 4  (* side-effect *);
        P1(B)      (* side-effect *);
        P2         (* side-effect *);
        P3         (* side-effect *);
        P4         (* side-effect *);
        RETURN 51
      END;

    BEGIN
      A := F(B);
    END;

  BEGIN
```

```
    P4
  END.
```

17.27 If you introduce a `FOR` loop into Clang (see Exercise 14.46), how could you prevent a malevolent program from altering the value of the loop control variable within the loop? Some attempts are easily detected, but those involving procedure calls are a little trickier, as study of the following might reveal:

```
PROGRAM Threaten;
  VAR i;

  PROCEDURE Nasty (VAR x);
    BEGIN
      x := 10
    END;

  PROCEDURE Nastier;
    BEGIN
      i := 10
    END;

  BEGIN
    FOR i := 0 TO 10 DO
      FOR i := 0 TO 5 DO (* Corrupt by using as inner control variable *)
        BEGIN
          READ(i)        (* Corrupt by reading a new value *);
          i := 6         (* Corrupt by direct assignment *);
          Nasty(i)       (* Corrupt by passing i by reference *);
          Nastier        (* Corrupt by calling a procedure having i in scope *)
        END
  END.
```

Further reading

Criticisms of well established languages like Pascal, Modula-2 and C are worth following up. The reader is directed to the classic papers by Welsh, Sneeringer and Hoare (1977) (reprinted in Barron (1981)), Kernighan (1981), Cailliau (1982), Cornelius (1988), Mody (1991), and Sakkinen (1992) for evidence that language design is something that does not always please users.

18 CONCURRENT PROGRAMMING

It is the objective of this chapter to extend the Clang language and its implementation to do what its name suggests - handle simple problems in concurrent programming. It is quite likely that this is a field which is new to the reader, and so we shall begin by discussing some rudimentary concepts in concurrent programming. Our treatment of this is necessarily brief, and the reader would be well advised to consult one of the excellent specialist textbooks for more detail.

18.1 Fundamental concepts

A common way of introducing programming to novices is by the preparation of a recipe or algorithm for some simple human activity, such as making a cup of tea, or running a bath. In such introductions the aim is usually to stress the idea of **sequential** algorithms, where "one thing gets done at a time". Although this approach is probably familiar by now to most readers, on reflection it may be seen as a little perverse to try to twist all problem solving into this mould - indeed, that may be the very reason why some beginners find the sequential algorithm a difficult concept to grasp. Many human activities are better represented as a set of interacting processes, which are carried out in parallel. To take a simple example, a sequential algorithm for changing a flat tyre might be written

```
begin
  open boot
  take jack from boot
  take tools from boot
  remove hubcap
  loosen wheel nuts
  jack up car
  take spare from boot
  take off flat tyre
  put spare on
  lower jack
  tighten wheel nuts
  replace hubcap
  place flat tyre in boot
  place jack in boot
  place tools in boot
  close boot
end
```

but it might be difficult to convince a beginner that the order here was correct, especially if he or she were used to changing tyres with the aid of a friend, when the algorithm might be better expressed

```
begin
                     open boot
   take tools from boot and     take jack from boot
                   remove hubcap
                 loosen wheel nuts
   jack up car           and     take spare from boot
                 take off flat tyre
                    put spare on
   lower jack            and     place flat tyre in boot
   tighten wheel nuts    and     place jack in boot
   replace hubcap        and     place tools in boot
                     close boot
end
```

Here we have several examples of **concurrent processes**, which could in theory be undertaken by two almost autonomous processors - provided that they co-operate at crucial instants so as to keep

in step (for example, taking off the flat tyre and getting the spare wheel from the boot are both processes which must be completed before the next process can start, but it does not matter which is completed first).

We shall define a **sequential process** as a sequence of operations carried out one at a time. The precise definition of an operation will depend on the level of detail at which the process is described. A **concurrent program** contains a set of such processes executing in parallel.

There are two motivations for the study of concurrency in programming languages. Firstly, concurrent facilities may be directly exploited in systems where one has genuine multiple processors, and such systems are becoming ever more common as technology improves. Secondly, concurrent programming facilities may allow some kinds of programs to be designed and structured more naturally in terms of autonomous (but almost invariably interacting) processes, even if these are executed on a single processing device, where their execution will, at best, be interleaved in **real time**.

Concurrent multiprocessing of peripheral devices has been common for many years, as part of highly specialized operating system design. Because this usually has to be ultra efficient, it has tended to be the domain of the highly skilled assembly-level programmer. It is only comparatively recently that high-level languages have approached the problem of providing reliable, easily understood constructs for concurrent programming. The modern programmer should have at least some elementary knowledge of the constructs, and of the main problem areas which arise in concurrent programming.

18.2 Parallel processes, exclusion and synchronization

We shall introduce the notation

$$\texttt{COBEGIN}\ S_1\ ;\ S_2\ ;\ S_3\ ;\ \ldots\ S_n\ \texttt{COEND}$$

to denote that the statements or procedure calls S_k can be executed concurrently. At an abstract level the programmer need not be concerned with how concurrency might be implemented at the hardware level on a computer - all that need be assumed is that processes execute in a non-negative, finite (as opposed to infinite) time. Whether this execution truly takes place in parallel, or whether it is interleaved in time (as it would have to be if only one processor was available) is irrelevant.

To define the effect of a concurrent statement we must take into account the statements S_0 and S_{n+1} which precede and follow it in a given program. The piece of code

$$S_0\ ;\ \texttt{COBEGIN}\ S_1\ ;\ S_2\ ;\ S_3\ ;\ \ldots\ S_n\ \texttt{COEND};\ S_{n+1}$$

can be represented by the **precedence graph** of Figure 18.1.

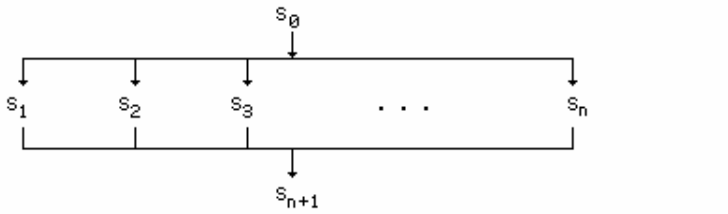


Figure 18.1 A precedence graph for a simple set of concurrent processes

Only after all the statements $S_1 \ldots S_n$ have been executed will S_{n+1} be executed. Similarly, the construction

```
S0;
COBEGIN
  S1;
  BEGIN
    S2; COBEGIN S3; S4 COEND; S5
  END;
  S6
COEND;
S7
```

can be represented by the precedence graph of Figure 18.2.

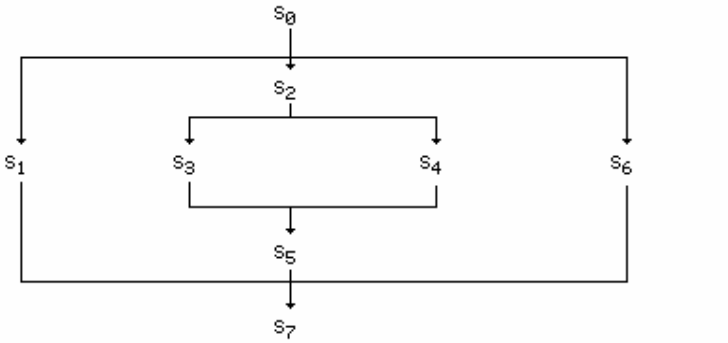


Figure 18.2 A precedence graph for a more complex set of processes

Although it is easy enough to depict code using the `COBEGIN ... COEND` construct in this way, we should observe that precedence graphs can be constructed which cannot be translated into this highly structured notation. An example of this appears in Figure 18.3.

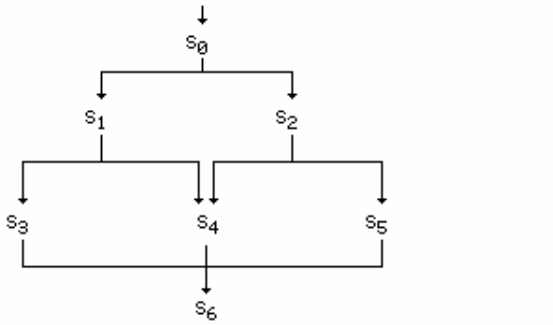


Figure 18.3 A precedence graph that cannot be expressed using COBEGIN ... COEND

As an example of the use of the `COBEGIN ... COEND` construct, we show a small program that will compute three simple summations simultaneously

```
PROGRAM Concurrent;
  VAR
    s1, s2, s3, i1, i2, i3;

  BEGIN
    COBEGIN
      BEGIN s1 := 0; FOR i1 := 1 TO 10 DO s1 := s1 + i1 END;
      BEGIN s2 := 0; FOR i2 := 1 TO 10 DO s2 := s2 + i2 END;
      BEGIN s3 := 0; FOR i3 := 1 TO 10 DO s3 := s3 + i3 END;
    COEND;
    WRITE(s1, s2, s3)
  END.
```

We may use this example to introduce two problem areas in concurrent programming that simply do not arise in sequential programming (at least, not that the high-level user can ever perceive). We have already intimated that we build concurrent programs out of sequential processes that must be regarded as executing simultaneously. A sequential process must be thought of as a sequential algorithm that operates on a data structure; the whole has the important property that it always gives the same result, regardless of how long it takes to execute. When sequential processes start to execute in parallel their time independence remains invariant only if their data structures remain truly private. If a process uses variables which other processes may simultaneously be changing, it is easy to see that the behaviour of the program as a whole may depend crucially on the relative speeds of each of its parts, and may become totally unpredictable.

In our example the three processes access totally private variables, so their concurrent composition is equivalent to any of the six possible ways in which they could have been arranged sequentially. As concurrent processes, however, the total execution time might be reduced. However, for the similar program below

```
PROGRAM Concurrent;
  VAR
    s1, s2, s3, i1, i2, i3;

  BEGIN
    COBEGIN
      BEGIN s2 := 0; FOR i1 := 1 TO 10 DO s1 := s1 + i2 END;
      BEGIN s3 := 0; FOR i2 := 1 TO 10 DO s2 := s2 + i3 END;
      BEGIN s1 := 0; FOR i3 := 1 TO 10 DO s3 := s3 + i1 END;
    COEND;
    Write(s1, s2, s3)
  END.
```

chaos would result, because we could never predict with certainty what was in any of the shared variables. At the same time the reader should appreciate that it must be possible to allow processes to access non-private data structures, otherwise concurrent processes could never exchange data and co-operate on tasks of mutual interest.

If one wishes to succeed in building large, reliable, concurrent programs, one will ideally want to use programming languages that cater specially for such problems, and are so designed that time dependent errors can be detected at compile-time, before they cause chaos - in effect the compiler must protect programmers from themselves. The simple `COBEGIN ... COEND` structure is inadequate as a reliable programming tool: it must be augmented with some restrictions on the forms of the statements which can be executed in parallel, and some method must be found of handling the following problems:

- **Communication** - processes must somehow be able to pass information from one to another.

- **Mutual exclusion** - processes must be guaranteed that they can access a critical region of code and/or a data structure in real-time without simultaneous interference from competing processes.

- **Synchronization** - two otherwise autonomous processes may have to be forced to wait in real-time for one another, or for some other event, and to signal one another when it is safe to proceed.

How best to handle these issues has been a matter for extensive research; suffice it to say that various models have been proposed and incorporated into modern languages such as Concurrent Pascal, Pascal-FC, Pascal-Plus, Modula, Edison, Concurrent Euclid, occam and Ada. Alarums and excursions: we propose to study a simple method, and to add it to Clang in this chapter.

We shall restrict the discussion to the use of shared memory for communication between processes (that is, processes communicate information through being able to access common areas of the same memory space, rather than by transmitting data down channels or other links).

The exclusion and synchronization problems, although fundamentally distinct, have a lot in common. A simple way of handling them is to use the concept of the **semaphore**, introduced by Dijkstra in 1968. Although there is a simple integer value associated with a semaphore `S`, it should really be thought of as a new type of variable, on which the only valid operations, beside the assignment of an initial associated integer value, are `P(S)` (from the Dutch *passeren*, meaning to pass) and `V(S)` (from the Dutch *vrijgeven*, meaning to release). In English these are often called **wait** and **signal**. The operations allow a process to cause itself to wait for a certain event, and then to be resumed when signalled by another process that the event has occurred. The simplest semantics of these operations are usually defined as follows:

> `P(S)` or `WAIT(S)` Wait until the value associated with `S` is positive, then subtract 1 from `S` and continue execution
>
> `V(S)` or `SIGNAL(S)` Add 1 to the value associated with `S`. This may allow a process that is waiting because it executed `P(S)` to continue.

Both `WAIT(S)` and `SIGNAL(S)` must be performed "indivisibly" - there can be no partial completion of the operation while something else is going on.

As an example of the use of semaphores to provide mutual exclusion (that is, protect a critical region), we give the following program, which also illustrates having two instances of the same process active at once.

```
PROGRAM Exclusion;
  VAR Shared, Semaphore;

  PROCEDURE Process (Limit);
    VAR Loop;
    BEGIN
      Loop := 1;
      WHILE Loop <= Limit DO
        BEGIN
          WAIT(Semaphore);
            Shared := Shared + 1;
          SIGNAL(Semaphore);
          Loop := Loop + 1;
        END
    END;

  BEGIN
    Semaphore := 1; Shared := 0;
    COBEGIN
      Process(4); Process(5+3)
    COEND;
    WRITE(Shared);
  END.
```

Each of the processes has its own private local loop counter `Loop`, but both increment the same

global variable `Shared`, access to which is controlled by the (shared) `Semaphore`. Notice that we are assuming that we can use a simple assignment to set an initial value for a semaphore, even though we have implied that it is not really a simple integer variable.

As an example of the use of semaphores to effect synchronization, we present a solution to a simple producer - consumer problem. The idea here is that one process produces items, and another consumes them, asynchronously. The items are passed through a distributor, who can only hold one item in stock at one time. This means that the producer may have to wait until the distributor is ready to accept an item, and the consumer may have to wait for the distributor to receive a consignment before an item can be supplied. An algorithm for doing this follows:

```
PROGRAM ProducerConsumer;
  VAR
    CanStore, CanTake;

  PROCEDURE Producer;
    BEGIN
      REPEAT
        ProduceItem;
        WAIT(CanStore); GiveToDistributor; SIGNAL(CanTake)
      FOREVER
    END;

  PROCEDURE Consumer;
    BEGIN
      REPEAT
        WAIT(CanTake); TakeFromDistributor; SIGNAL(CanStore);
        ConsumeItem
      FOREVER
    END;

  BEGIN
    CanStore := 1;  CanTake := 0;
    COBEGIN
      Producer; Consumer
    COEND
  END.
```

A problem which may not be immediately apparent is that communicating processes which have to synchronize, or ensure that they have exclusive access to a critical region, may become **deadlocked** when they all - perhaps erroneously - end up waiting on the same semaphore (or even different ones), with no process still active which can signal others. In the following variation on the above example this is quite obvious, but it is not always so simple to detect deadlock, even in quite simple programs.

```
PROGRAM ProducerConsumer;
  VAR
    CanStore, CanTake;

  PROCEDURE Producer (Quota);
    VAR I;
    BEGIN
      I := 1;
      WHILE I <= Quota DO
        BEGIN
          ProduceItem; I := I + 1;
          WAIT(CanStore); GiveToDistributor; SIGNAL(CanTake);
        END
    END;

  PROCEDURE Consumer (Demand);
    VAR I;
    BEGIN
      I := 1;
      WHILE I <= Demand DO
        BEGIN
          WAIT(CanTake); TakeFromDistributor; SIGNAL(CanStore);
          ConsumeItem; I := I + 1;
        END
    END;

  BEGIN
```

```
  CanStore := 1;  CanTake := 0;
  COBEGIN
    Producer(12); Consumer(5)
  COEND
END.
```

Here the obvious outcome is that only the first five of the objects produced can be consumed - when `Consumer` finishes, `Producer` will find itself waiting forever for the `Distributor` to dispose of the sixth item.

In the next section we shall show how we might implement concurrency using `COBEGIN ... COEND`, and the `WAIT` and `SIGNAL` primitives, by making additions to our simple language like those suggested above. This is remarkably easy to do, so far as compilation is concerned. Concurrent execution of the programs so compiled is another matter, of course, but we shall suggest how an interpretive system can give the effect of simulating concurrent execution, using run-time support rather like that found in some real-time systems.

Exercises

18.1 One of the classic problems used to illustrate the use of semaphores is the so-called "bounded buffer" problem. This is an enhancement of the example used before, but where the distributor can store up to `Max` items at one time. In computer terms these are usually held in a circular buffer, stored in a linear array, and managed by using two indices, say `Head` and `Tail`. In terms of our simple language we should have something like

```
CONST
  Max = Size of Buffer;
VAR
  Buffer[Max-1], Head, Tail;
```

with `Head` and `Tail` both initially set to 1. Adding to the buffer is always done at the tail, and removing from the buffer is done from the head, along the lines of

```
add to buffer:
    Buffer[Tail] := Item;
    Tail := (Tail + 1) MOD Max;

remove from buffer:
    Item := Buffer[Head];
    Head := (Head + 1) MOD Max;
```

Devise a system where one process continually adds to the buffer, at the same time that a parallel process tries to empty it, with the restrictions that (a) the first process cannot add to the buffer if it is full (b) the second process cannot draw from the buffer if it is empty (c) the first process cannot add to the buffer while the second process draws from the buffer at exactly the same instant in real-time.

18.2 Another classic problem has become known as Conway's problem, after the person who first proposed it. Write a program to read the data from 10 column cards, and rewrite it in 15 column lines, with the following changes: after every card image an extra space character is appended, and every adjacent pair of asterisks is replaced by a single up-arrow ↑.

This is easily solved by a single sequential program, but may be solved (more naturally?) by three concurrent processes. One of these, `Input`, reads the cards and simply passes the characters (with the additional trailing space) through a finite buffer, say `InBuffer`, to a process `Squash` which simply looks for double asterisks and passes a stream of modified characters through a second finite

buffer, say `OutBuffer`, to a process `Output`, which extracts the characters from the second buffer and prints them in 15 column lines.

18.3 A semaphore-based system - syntax, semantics, and code generation

So as to provide a system with which the reader can experiment in concurrent programming, we shall add a few more permissible statements to our language, as described by the following EBNF:

```
Statement           =   [ CompoundStatement | Assignment | ProcedureCall
                        | IfStatement | WhileStatement
                        | WriteStatement | ReadStatement
                        | CobeginStatement | SemaphoreStatement ] .
CobeginStatement    =   "COBEGIN" ProcessCall { ";" ProcessCall } "COEND" .
ProcessCall         =   ProcIdentifier ActualParameters .
SemaphoreStatement  =   ( "WAIT" | "SIGNAL" ) "(" Variable ")" .
```

There is no real restriction in limiting the statements which may be processed concurrently to procedure calls, as any other statement may be packaged into a trivial procedure. However, for our simple implementation we shall limit the number of processes which can execute in parallel, and restrict the `COBEGIN ... COEND` construction to appearing in the main program block. These restrictions are really imposed by our pseudo-machine, which we shall augment as simply as possible by incorporating five new instructions, `CBG, FRK, CND, WGT` and `SIG`, with the following semantics:

`CBG N`	Prepare to instantiate a set of `N` concurrent processes
`FRK A`	Set up a suspended call to a level 1 procedure whose code commences at address `A`
`CND`	Suspend parent process and transfer control to one of the processes previously instantiated by `FRK`
`WGT`	Wait on the semaphore whose address is at top-of-stack
`SIG`	Signal the semaphore whose address is at top-of-stack.

The way in which parsing and code generation are accomplished can be understood with reference to the Cocol extract that follows:

```
CobeginStatement
=                           (. int processes = 0;
                               CGEN_labels start; .)
   "COBEGIN"                (. if (blockclass != TABLE_progs) SemError(215);
                               CGen->cobegin(start); .)
   ProcessCall              (. processes++; .)
   { WEAK ";" ProcessCall   (. processes++; .)
   }
   "COEND"                  (. CGen->coend(start, processes); .) .

ProcessCall
=                           (. TABLE_entries entry; TABLE_alfa name;
                               bool found; .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (entry.idclass == TABLE_procs)
                                 CGen->markstack();
                               else { SemError(217); return; } .)
   ActualParameters<entry> (. CGen->forkprocess(entry.p.entrypoint); .) .

SemaphoreStatement
=                           (. bool wait; .)
   (  "WAIT"                (. wait = true; .)
    | "SIGNAL"              (. wait = false; .)
   )
   "(" Variable             (. if (wait) CGen->waitop(); else CGen->signalop(); .)
```

```
")" .
```

The reader should note that:

- Code associated with `COBEGIN` and `COEND` must be generated so that the run-time system can prepare to schedule the correct number of processes. A count of the processes is maintained in the parser for *CobeginStatement*, and later patched into the code generated by the call to the `cobegin` code generating routine when the call to `coend` is made.

- The `forkprocess` routine generates code that resembles a procedure call, but when this code is later executed it stops short of making the calls. In a more sophisticated code generator employing the use of an AST, as was discussed in section 17.6.2, the `PROCNODE` class would need to incorporate two code generating members, one for normal calls, and one for such suspended calls.

- The code generated by the `coend` routine signals to the run-time system that all the processes that have been initiated by the code generated by `forkprocess` may actually commence concurrent execution, and at the same time suspends operation of the main program. This will become clearer in the next section, where we discuss run-time support.

As before, the discussion will be clarified by presenting the code for an earlier example, which shows the use of semaphores to protect a critical region of code accessing a shared variable, the use of processes that use simple value parameters, and the invocation of more than one instance of the same process.

```
Clang 4.0 on 28/12/95 at 15:27:13

  0 PROGRAM Exclusion;
  0   VAR Shared, Semaphore;
  2
  2   PROCEDURE Process (Limit);
  2     VAR Loop;
  4     BEGIN
  4       Loop := 1;
 10       WHILE Loop <= Limit DO
 21         BEGIN
 21           WAIT(Semaphore);
 25             Shared := Shared + 1;
 36           SIGNAL(Semaphore);
 40           Loop := Loop + 1;
 51         END
 51     END;
 56
 56   BEGIN
 58     Semaphore := 1; Shared := 0;
 70     COBEGIN
 70       Process(4); Process(5+3)
 83     COEND;
 86     WRITE(Shared);
 92   END.
```

The code produced in the compilation of this program would read

```
  0 BRN       56    to start of main program
  2 DSP        1  BEGIN Process
  4 ADR  2    -6      address of Loop
  7 LIT        1      Constant 1
  9 STO             Loop := 1
 10 ADR  2    -6      address of Loop
 13 VAL               value of Loop
 14 ADR  2    -5      address of Limit
 17 VAL               value of Limit
 18 LEQ               Loop <= Limit ?
 19 BZE       53    WHILE Loop <= Limit DO
 21 ADR  1    -2        Address of Semaphore
 24 WGT               WAIT(Semaphore)
 25 ADR  1    -1        Address of Shared
```

```
 28 ADR  1    -1        Address of Shared
 31 VAL                 Value of Shared
 32 LIT        1        Constant 1
 34 ADD                 Value of Shared + 1
 35 STO               Shared := Shared + 1
 36 ADR  1    -2        Address of Semaphore
 39 SIG                 SIGNAL(Semaphore)
 40 ADR  2    -6        Address of Loop
 43 ADR  2    -6        Address of Loop
 46 VAL                 Value of Loop
 47 LIT        1        Constant 1
 49 ADD                 Value of Loop + 1
 50 STO               Loop := Loop + 1
 51 BRN       10    END
 53 RET  2     0  END Process
 56 DSP        2  BEGIN Exclusion
 58 ADR  1    -2      Address of Semaphore
 61 LIT        1      Constant 1
 63 STO              Semaphore := 1
 64 ADR  1    -1      Address of Shared
 67 LIT        0      Constant 0
 69 STO              Shared := 0
 70 CBG        2     COBEGIN (2 processes)
 72 MST                  Mark stack
 73 LIT        4         Argument 4
 75 FRK        2       Process(4)
 77 MST                  Mark Stack
 78 LIT        5         Constant 5
 80 LIT        3         Constant 3
 82 ADD                  Argument 5 + 3
 83 FRK        2       Process(5+3)
 85 CND              COEND
 86 ADR  1    -1      Address of Shared
 89 VAL                Value of Shared
 90 PRN              WRITE(Shared)
 91 NLN              WriteLn
 92 HLT            END Exclusion
```

Exercises

18.3 If you study the above code carefully you might come up with the idea that it could be optimized by adding "level" and "offset" components to the `WGT` and `SIG` instructions. Is this a feasible proposition?

18.4 What possible outputs would you expect from the example program given here? What outputs could you expect if the semaphore were not used?

18.5 Is it not a better idea to introduce `PROCESS` as a reserved keyword, rather than just specifying a process as a `PROCEDURE`? Discuss arguments for and against this proposal, and try to implement it anyway.

18.4 Run-time implementation

We must now give some consideration to the problem of how one might execute a set of parallel processes or, in our case, interpret the stack machine code generated by the compiler. Perhaps this is a good point to comment that any sequential process forming part of a (generally) concurrent program may be in one of four states:

- *running* - instructions are being executed

- *ready* - suspended, waiting to be assigned to a processor

- *blocked* - suspended, waiting for some event to occur (such as I/O to be completed, or a signal on a semaphore)

- *deadlocked* - suspended, waiting for an event that will never occur (perhaps because of a failure in some other part of the system).

These states may conveniently be represented on a state diagram like that of Figure 18.4.

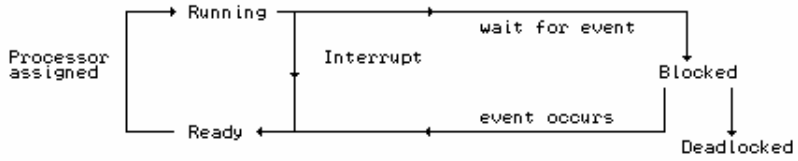


Figure 18.4 The possible execution states of one of a number of sequential processes

In practical implementations, concurrent behaviour is achieved in one of several ways. For example, processes can either

- share execution time on a single real processor (pseudo-concurrency);
- execute on a true multiprocessor system with shared memory, perhaps with each processor handling at most one process;
- execute on a true multiprocessor system without shared memory (distributed processing).

The implementation usually depends critically on a run-time support system or **kernel**, which may take one of a number of forms:

- a software structure, programmed as part of the application (as must be done in Modula-2);
- a standard software system, linked in with the object code (as usually done in Ada);
- a microcoded hardware structure (as typified by the Transputer).

Although the *logical behaviour* of a correct concurrent program will not - or should not - be dependent on the kernel, the *performance* of a real-time system may depend critically on the characteristics of the scheduling algorithms used in the kernel.

The shared memory, semaphore-based implementation upon which we have been focusing attention lends itself to the idea of multiplexing the processes on a single processor, or distributing them among a set of processors. Our interpreter, of course, really runs on one processor, although there is no reason why it should not emulate several real processors - with an interpretive approach any architecture can be modelled if one is prepared to sacrifice efficiency. What we shall do here is to emulate a system in which one controlling processor shares its time between several processes, allowing each process to execute for a few simulated fetch-execute cycles, before moving on to the next. This idea of **time-slicing** is very close to what occurs in some time-sharing systems in real life, with one major difference. Real systems are usually interrupt-driven by clock and peripheral controller devices, with hardware mechanisms controlling when some process switches occur, and software mechanisms controlling when others happen as a result of `WAIT` and `SIGNAL` operations on semaphores. On our toy system we shall simulate time-slicing by letting each active process

execute for a small random number of steps before control is passed to another one.

The simulated shared memory of the complete system will be divided up between the parallel processes while they are executing. This is not the place to enter into a study of sophisticated memory management techniques. Instead, what we shall do is to divide the memory which remains after the allocation to the main program stack frame, the program code, and the string pool, equally among each of the processes which have been initiated.

The processes are started by the main program; while they are executing, the main program is effectively dormant. When all the processes have run to completion, the main program is activated once more. While they are running, one can think of each as a separate program, each requiring its own stack memory, and each managing it in the way discussed previously. Each process conceptually has its own processor - or, more honestly, keeps track of its own set of processor registers, its own display, and so on. To accomplish this, we extend the data structures used by the interpreter, and in particular introduce a linked ring structure of so-called **process descriptors**, as follows:

```
const int STKMC_procmax = 10;          // Limit on concurrent processes
typedef int STKMC_procindex;           // Really 0 .. procmax

struct processrec {                    // Process descriptor records
  STKMC_address bp, mp, sp, pc;        // Shadow registers
  STKMC_procindex next;                // Ring pointer
  STKMC_procindex queue;               // Linked, waiting on semaphore
  bool ready;                          // Process ready flag
  STKMC_address stackmax, stackmin;    // Memory limits
  int display[STKMC_levmax];           // Display registers
};

processrec process[STKMC_procmax + 1]; // Ring of process descriptors
STKMC_procindex current, nexttorun;    // Process pointers
const int maxslice = 8;                // Maximum time slice
int slice;                             // Current time slice
```

The reader should note the following:

- The important `ready` field indicates whether the process is still active (`ready = true`), or has run to completion or been suspended on a semaphore (`ready = false`).
- Each process descriptor needs to maintain copies of the processor registers, so that when a **context switch** is done to allow another process to take charge of the single processor, the real CPU registers can be restored to the values they had when that process last executed.
- Similarly, each process descriptor maintains its own set of display registers, and its own limits on the memory that it is allowed to access for storing local variables and performing stack manipulations.
- The zeroth entry in the `process` array is used for the main program. As already mentioned, the other process descriptors are linked to form a circular ring, and the `next` and `queue` fields are used to connect these descriptors together.

As an example, consider the case where the main program has just launched four concurrent processes. The process descriptors would be linked as shown in Figure 18.5(a).

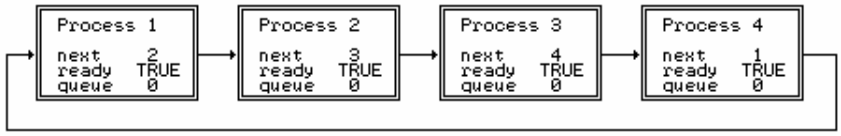


Figure 18.5(a) Process ring immediately after launching four concurrent processes

If process 2 is then forced to wait on a semaphore, the descriptor ring would change to the situation depicted in Figure 18.5(b).

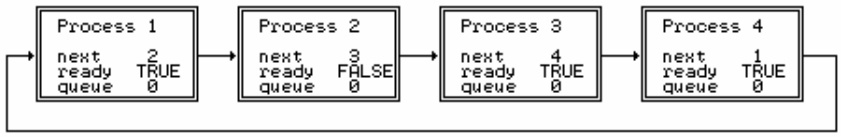


Figure 18.5(b) Process ring immediately after process 2 has been forced to wait

If process 3 runs to completion in the next time slice, the ring will then change to the situation depicted in Figure 18.5(c).

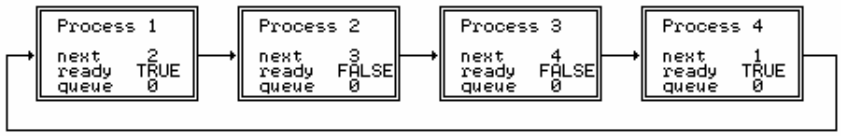


Figure 18.5(c) Process ring immediately after process 3 has run to completion

Finally, if process 4 then waits on the same semaphore, the ring changes to the situation depicted in Figure 18.5(d).

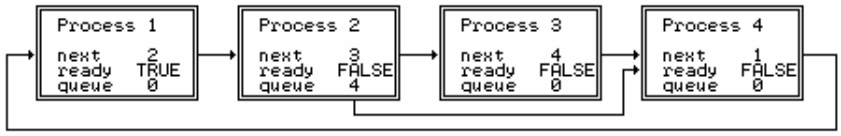


Figure 18.5(d) Process ring after process 4 has also been forced to wait

When a group of processes are all waiting on a common semaphore, their `ready` fields will all have been set to `false`, and their `queue` fields will have been used to link them in a FIFO queue, set up in real time as the `WAIT` operations were handled. We return to this point a little later on.

Initialization and emulation of the machine proceeds much as before, save that we now initialize a parent process (main program) process descriptor as well as the virtual processor:

```
process[0].queue = 0;           // Initialize parent process descriptor
process[0].ready = true;        // (memory limits and display)
process[0].stackmax = initsp;
process[0].stackmin = codelen;
for (int l = 0; l < STKMC_levmax; l++) process[0].display[l] = initsp;
cpu.sp = initsp;                // Initialize stack pointer
cpu.bp = initsp;                // Initialize registers
cpu.pc = initpc;                // Initialize program counter
nexttorun = 0; nprocs = 0;      // Initialize emulator variables
slice = 0; ps = running;
do
```

```
  { current = nexttorun;          // Set active process descriptor pointer
    pcnow = cpu.pc;               // Save for tracing purposes
    if (unsigned(mem[cpu.pc]) > int(STKMC_nul)) ps = badop;
    else
    { cpu.ir = STKMC_opcodes(mem[cpu.pc]); cpu.pc++;  // Fetch
      if (tracing) trace(results, pcnow);
      switch (cpu.ir) {                               // Execute
                                                      // various cases
      }
    }
    if (nexttorun != 0) chooseprocess();
  } while (ps == running);
```

As we shall see later, `display[0]` is set to `initsp` for all processes, and will not change, for all processes are able to access the global variables of the main program. This is the most effective means we have of sharing data between processes.

Two pointers are used to index the array of process descriptors. `nexttorun` indicates the process that has most recently been *assigned* to the processor, and `current` indicates the process that is currently *running*. Each iteration of the fetch-execute cycle begins by copying `nexttorun` to `current`; some operations will alter the value of `nexttorun` to indicate that the real processor should be assigned to a new process. In particular, once the concurrent processes begin execution, `nexttorun` will no longer have the value of zero. The last part of the processing loop detects this as an indication that it may have to choose another process.

The algorithm for `chooseprocess` makes use of the variable `slice`. This is set to a small random number at the start of concurrent processing, and thereafter is decremented after each pseudo-machine instruction, or set to zero when a process is forced to wait, or terminates normally. When `slice` reaches zero, the process descriptor ring is searched cyclically (using the `next` pointer) so as to find a suitable process with which to continue for a further small (random) number of steps. Once found, a **context switch** is performed - the current CPU registers must be saved in the process descriptor, and must then be replaced by the values apposite to the process that is about to continue. The search and context switch are easily programmed:

```
void STKMC::swapregisters(void)
// Save current machine registers; restore from next process
{ process[current].bp = cpu.bp;   cpu.bp = process[nexttorun].bp;
  process[current].mp = cpu.mp;   cpu.mp = process[nexttorun].mp;
  process[current].sp = cpu.sp;   cpu.sp = process[nexttorun].sp;
  process[current].pc = cpu.pc;   cpu.pc = process[nexttorun].pc;
}

void STKMC::chooseprocess(void)
// From current process, traverse ring of descriptors to next ready process
{ if (slice != 0) { slice--; return; }
  do { nexttorun = process[nexttorun].next; }
  while (!process[nexttorun].ready);
  if (nexttorun != current) swapregisters();
  slice = random(maxslice) + 3;
}
```

We are here presuming that we have a suitable library function `random(limit)` for generating a sequence of random numbers, suitably scaled to lie in the range `0 <= random < Limit`.

There is a point of some subtlety here. If the search is instigated by virtue of one process being forced to wait on a semaphore or terminating normally, it must find *another* process to execute. There may be no such process, in which case a state of deadlock can be detected. However, if the search is instigated simply by virtue of a process reaching the end of its allotted time slice, then control can legitimately return to the same process if no other ready process can be found.

The mechanism of the `COBEGIN ... COEND` system is next to be discussed. As we have extended the language, processes are syntactically indistinguishable from procedures, and the code generation between the `COBEGIN` and `COEND` very nearly, but not quite, generates a set of procedure

calls. There is a fundamental difference, of course, in the way in which such procedure "calls" execute. After the `COBEGIN`, transfer of control must not pass immediately to the process procedures, but must remain with the main parent program until all child processes can be started together - the reason being that parameters may have to be set up, and this will have to be done in the environment of the parent.

For our stack machine, the code generated by the `cobegin` routine (in our simple machine, the `CBG N` sequence) is used by the kernel to decide on how to divide the remaining memory up among the imminent processes. This is achieved by the following code in the emulator:

```
case STKMC_cbg:
  if (mem[cpu.pc] > 0)                              // any processes?
  { partition = (cpu.sp - codelen) / mem[cpu.pc];  // divide rest of memory
    parentsp = cpu.sp;                              // for restoration by cnd
  }
  cpu.pc++;
  break;
```

The necessity of remembering the current value of `cpu.sp` will immediately become apparent after studying the interpretation of the `FRK A` instruction which is executed in place of the rather similar `CAL L A` so as to set up a process. Essentially what has to be achieved is the setting up of a complete activation record and process descriptor for a procedure, but without transferring control to this:

```
case STKMC_frk:
  nprocs++;                                       // one more process
  // first initialize the shadow CPU registers and display
  process[nprocs].bp = cpu.mp;                    // base pointer
  process[nprocs].mp = cpu.mp;                    // mark stack pointer
  process[nprocs].sp = cpu.sp;                    // stack pointer
  process[nprocs].pc = mem[cpu.pc];               // process entry point
  process[nprocs].display[0] =
    process[0].display[0];                        // for global access
  process[nprocs].display[1] = cpu.mp;            // for local access
  // now initialize frame header
  mem[process[nprocs].bp - 2] =
    process[0].display[1];                        // display copy
  mem[process[nprocs].bp - 3] = cpu.bp;           // dynamic link
  mem[process[nprocs].bp - 4] = processreturn;    // return address
  // descriptor house keeping
  process[nprocs].stackmax = cpu.mp;              // memory limits
  process[nprocs].stackmin = cpu.mp - partition;
  process[nprocs].ready = true;                   // ready to run
  process[nprocs].queue = 0;                      // clear semaphore queue
  process[nprocs].next = nprocs + 1;              // link to next descriptor
  cpu.sp = cpu.mp - partition;                    // bump parent sp below
  cpu.pc++;                                       // memory reserved for process
  break;
```

where the reader should note that:

- The return address for a process procedure is set to an artificial value (this might be zero, but any other "impossible" value would suffice). This can later be detected at procedure exit as an indication that the process is complete, and may be deactivated.

- The penultimate step involves resetting the stack pointer for the parent process so as to skip over the area in memory that is being reserved for the process workspace.

The mechanics of `COEND` are now easy: we merely deactivate the main program, close the descriptor ring, and choose one of the processes (at random) to continue execution. When all processes have run to completion their workspaces can, of course, all be reclaimed. Provision for doing this was made when we saved the value of `cpu.sp` as part of the action of the `CBG N` instruction; the saved value is restored to the process descriptor for the main program as part of the interpretation of the `CND` instruction:

```
case STKMC_cnd:
  if (nprocs > 0)                        // check for pathological case
  { process[nprocs].next = 1;            // close ring of descriptors
    nexttorun = random(nprocs) + 1;      // choose first process at random
    cpu.sp = parentsp;                   // restore parent stack pointer
  }
  break;
```

Processes, like procedures, terminate when they encounter a `RET` instruction. The interpretation requires slight modification from what we have seen previously, and may be understood with reference to the code below:

```
case STKMC_ret:
  process[current].display[mem[cpu.pc] - 1] = mem[cpu.bp - 2];
                                         // restore display
  cpu.sp = cpu.bp - mem[cpu.pc + 1];     // discard stack frame
  cpu.mp = mem[cpu.bp - 5];              // restore mark pointer
  cpu.pc = mem[cpu.bp - 4];              // get return address
  cpu.bp = mem[cpu.bp - 3];              // reset base pointer
  if (cpu.pc == processreturn)           // kill a concurrent process
  { nprocs--; slice = 0;                 // force choice of new process
    if (nprocs == 0)                     // must reactivate main program
    { nexttorun = 0; swapregisters(); }
    else                                 // complete this process only
    { chooseprocess();                   // may fail
      process[current].ready = false;
      if (current == nexttorun) ps = deadlock;
    }
  }
  break;
```

Much of this is as before, except that we must check for the artificial return address mentioned above. If this is detected, but uncompleted processes are known to exist, we reset the time slice, attempt to choose another process, switch context, deactivate the completed process, and only then check for deadlock. On the other hand, when all processes have been completed, we simply do a context switch back to the main program (`process[0]`).

The last point to be considered is that of implementing semaphore operations. This is a little subtle. The simplest semantic meaning for the `WAIT` and `SIGNAL` operations is probably

```
WAIT(S)       WHILE S < 0 DO (* nothing *) END; S := S - 1;

SIGNAL(S)     S := S + 1;
```

where, as we have remarked, the testing and incrementing must be be done as indivisible operations. The interpreter allows easy implementation of this otherwise rather awkward property, because the entire operation can be handled by one pseudo-operation (a `WGT` or `SIG` instruction).

However, the simple semantic interpretation above is probably never implemented, for it implies what is known as a **busy-wait** operation, where a processor is tied up cycling around wasting effort doing nothing. Implementations of semaphores prefer to deactivate the waiting process completely, possibly adding it to a queue of such processes, which may later be examined efficiently when a signal operation gives the all-clear for a process to continue. Although the semantics of `SIGNAL` do not require a queue to be formed, we have chosen to employ one here.

The `WAIT` and `SIGNAL` primitives can then be implemented in several ways. For example, `WAIT(Semaphore)` can be realized with an algorithm like

```
IF Semaphore.Count > 0
  THEN DEC(Semaphore.Count)
  ELSE set Slice to 0 and ChooseProcess
       Process[Current].Ready := FALSE
       add Process[Current] to Semaphore.Queue
       Process[Current].Queue := 0
END
```

provided that the matching `SIGNAL(Semaphore)` is realized by an algorithm like

```
IF Semaphore.Queue is empty
  THEN INC(Semaphore.Count)
  ELSE find which process should be Woken
       Process[Woken].Ready := TRUE
       set start of Semaphore.Queue to point to Process[Woken].Queue
END
```

The problem then arises of how to represent a semaphore variable. The first idea that might come to mind is to use something on the lines of a structure or record with two fields, but this would be awkward, as we should have to introduce further complications into the parser to treat variables of different sizes. We can retain simplicity by noting that we can use an integer to represent a semaphore if we allow negative values to act as `Queue` values and non-negative values to act as `Count` values. With this idea we simply modify the interpreter to read

```
case STKMC_wgt:
  if (current == 0) ps = badsem;
  else { cpu.sp++; wait(mem[cpu.sp - 1]); }
  break;
case STKMC_sig:
  if (current == 0) ps = badsem;
  else { cpu.sp++; signal(mem[cpu.sp - 1]); }
  break;
```

with `wait` and `signal` as routines private to the interpreter, defined as follows:

```
void STKMC::signal(STKMC_address semaddress)
{ if (mem[semaddress] >= 0)                 // do we need to waken a process?
  { mem[semaddress]++; return; }            // no - simply increment semaphore
  STKMC_procindex woken = -mem[semaddress]; // negate to find index
  mem[semaddress] = -process[woken].queue;  // bump queue pointer
  process[woken].queue = 0;                 // remove from queue
  process[woken].ready = true;              // and allow to be reactivated
}

void STKMC::wait(STKMC_address semaddress)
{ STKMC_procindex last, now;
  if (mem[semaddress] > 0)                  // do we need to suspend?
  { mem[semaddress]--; return; }            // no - simply decrement semaphore
  slice = 0; chooseprocess();               // choose the next process
  process[current].ready = false;           // and suspend this one
  if (current == nexttorun) { ps = deadlock; return; }
  now = -mem[semaddress];                   // look for end of semaphore queue
  while (now != 0) { last = now; now = process[now].queue; }
  if (mem[semaddress] == 0)
    mem[semaddress] = -current;             // first in queue
  else
    process[last].queue = current;          // place at end of existing queue
  process[current].queue = 0;               // and mark as the new end of queue
}
```

There are, as always, some subtleties to draw to the reader’s attention:

- A check should be made to see that `WAIT` and `SIGNAL` are only attempted from within a concurrent process. Because of the way in which we have extended the language, with processes being lexically indistinguishable from other procedures, this cannot readily be detected at compile-time, but has to be done at run-time. (See also Exercise 18.5.)

- Although the semantic definition above also seems to imply that the value of a semaphore is always increased by a `SIGNAL` operation, we have chosen not to do this if a process is found waiting on that semaphore. This process, when awoken from its implied busy-wait loop, would simply decrement the semaphore anyway; there is no need to alter it twice.

- The semantics of `SIGNAL` do not require that a process which is allowed to proceed actually gain control of the processor immediately, and we have not implemented `signal` in this way.

Exercises

18.6 Add concurrent processing facilities to Topsy on the lines of those described here.

18.7 Introduce a call to a random number generator as part of Clang or Topsy (as a possibility for a *Factor*), which will allow you to write simple simulation programs.

18.8 Ben-Ari (1982) and Burns and Davies (1993) make use of a `REPEAT - FOREVER` construct. How easy is it to add this to our language? How useful is it on its own?

18.9 A multi-tasking system can easily devote a considerable part of its resources to process switching and housekeeping. Try to identify potential sources of inefficiency in our system, and eradicate as many as possible.

18.10 One problem with running programs on this system is that in general the sequence of interleaving the processes is unpredictable. While this makes for a useful simulation in many cases, it can be awkward to debug programs which behave in this way, especially with respect to I/O (where individual elements in a read or write list may be separated by elements from another list in another process). It is easy to use a programmer-defined semaphore to prevent this; can you also find a way of ensuring that process switching is suspended during I/O, perhaps requested by a compiler directive, such as `(*$S-*)`?

18.11 Is it difficult to allow concurrent processes to be initiated from within procedures and/or other processes, rather than from the main program only? How does this relate to Exercise 18.5?

18.12 Develop an emulation of a multiprocessor system. Rather than have only one processor, consider having an (emulated) processor for each process.

18.13 Remove the restriction on a fixed upper limit to the number of processes that can be accommodated, by making use of process descriptors that are allocated dynamically.

18.14 Our round-robin scheduler attempts to be fair by allocating processor time to each ready process in rotation. Develop a version that is unfair, in that a process will only relinquish control of the processor when it is forced to wait on a semaphore, or when it completes execution. Is this behaviour typical of any real-time systems in practice?

18.15 As an extension to Exercise 18.14, implement a means whereby a process can voluntarily suspend its own action and allow another process of the same or higher priority to assume charge of the processor, perhaps by means of a routine `SUSPEND`.

18.16 Do you suppose that when a process is signalled it should be given immediate access to the processor? What are the implications of allowing or disallowing this strategy? How could it be implemented in our system?

18.17 Replace the kernel with one in which semaphores do not have an associated queue. That is, when a `SIGNAL(S)` operation finds one or more processes waiting on `S`, simply choose one of these processes at random to make `ready` again.

18.18 Our idea of simply letting a programmer treat a semaphore as though it were an integer is

scarcely in the best traditions of strongly typed languages. How would you introduce a special semaphore type into Clang or Topsy (allow for arrays of semaphores), and how would you prevent programmers from tampering with them, while still allowing them to assign initial values to their `Count` fields? You might like to consider other restrictions on the use of semaphores, such as allowing initial assignment only within the parent process (main program), forbidding assignment of negative values, and restricting the way in which they can be used as parameters to procedures, functions or processes (you will need to think very carefully about this).

18.19 In our system, if one process executes a `READ` operation, the whole system will wait for this to be completed. Can you think of a way in which you can prevent this, for example by checking to see whether a key has been pressed, or by making use of real-time interrupts? As a rather challenging exercise, see if you can incorporate a mechanism into the interpreter to provide for so-called *asynchronous input*.

18.20 The idea of simulating time-slicing by allowing processes to execute for a small random number of steps has been found to be an excellent teaching tool (primarily because subtly wrong programs often show up faults very quickly, since the scheduler is essentially non-deterministic). However, real-time systems usually implement time-slicing by relying on interrupts from a real-time clock to force context switches at regular intervals. A Modula-2 implementation of an interpreter can readily be modified to work in this way, by making use of coroutines and the `IOTRANSFER` procedure. As a rather challenging exercise, implement such an interpreter. It is inexpedient to implement true time-slicing - pseudo-code operations (like `WGT` and `SIG`) should remain indivisible. A suggested strategy to adopt is one where a real clock interrupt sets a flag that the repetitive fetch-execute cycle of the emulator can acknowledge; furthermore, it might be advantageous to slow the real rate of interrupts down to compensate for the fact that an interpreter is far slower than a "real" computer would be.

Many kernels employ, not a ring of process descriptors, but one or more prioritized queues. One of these is the **ready queue**, whose nodes correspond to processes that are ready to execute. The process at the front of this queue is activated; when a context switch occurs the descriptor is either moved to another queue (when the process is forced to wait on a semaphore), is deallocated (when the process has finished executing), or is re- queued in the ready queue behind other processes of equal priority (if fair scheduling is employed on a round- robin basis). This is a method that allows for the concept of processes to be assigned relative priorities rather more easily than if one uses a ring structure. It also gives rise to a host of possibilities for redesigning the kernel and the language.

18.21 Develop a kernel in which the process descriptors for ready processes (all of the same priority) are linked in a simple ready queue. When the active process is forced to wait on a semaphore, transfer its descriptor to the appropriate semaphore queue; when the active process reaches the end of its time slice, transfer its descriptor to the end of the ready queue. Does this system have any advantages over the ring structure we have demonstrated in this section?

18.22 Extend the language and the implementation so that processes may be prioritized. When a context switch occurs, the scheduler always chooses the ready process with the highest priority (that is, the one at the front of the queue) as the one to execute next. There are various ways in which process priority might be set or changed. For example:

- Develop a system where process priorities are determined at the time the processes are spawned, and remain constant thereafter. This could be done by changing the syntax of the `COBEGIN ... COEND` structure:

```
CobeginStatement =  "COBEGIN" ProcessCall { ";" ProcessCall } "COEND" .
```

```
ProcessCall      = ProcIdentifier ActualParameters [ Priority ] .
Priority         = "[" Expression "]" .
```

Take care to ensure that the *Expression* is evaluated and stored in the correct context.

- Develop a system where processes may alter their priorities as they execute, say by calling on a routine `SETPRIORITY(Priority)`.

Pay particular attention to the way in which semaphore queues are manipulated. Should these be prioritized in the same way, or should they remain FIFO queues?

Further reading

Several texts provide descriptions of run-time mechanisms rather similar to the one discussed in this chapter.

In Ben-Ari's influential book *Principles of Concurrent Programming* (1982) may be found an interpreter for a language based on Pascal-S (Wirth, 1981). This implementation has inspired several others (including our own), and also formed the starting point for the Pascal-FC implementation described by Burns and Davies in their excellent and comprehensive book (1993). Burns and Davies also outline the implementation of the support for several other concurrent paradigms allowed by their language.

We should warn the reader that our treatment of concurrent programming, like that of so much else, has been rather dangerously superficial. He or she might do well to consult one or more of the excellent texts which have appeared on this subject in recent years. Besides those just mentioned, we can recommend the books by Burns and Welling (1989), and Bustard, Elder and Welsh (1988) and the survey paper by Andrews and Schneider (1983).

Appendix A

Software resources for this book

(This appendix is a considerably modified version of the one that appeared in the printed book. For interest, the original one can be found in the file appa0.pdf.)

A.1 Source code for the programs

The software that accompanies this text was originally developed in Modula-2. It was subsequently converted to Turbo Pascal, and to C++. Although C++ code is used for most of the illustrations in the text, highly self-consistent source code in all three languages is to be found in this distribution, along with language-specific implementation notes.

The C++ source code was mainly developed under MS-DOS using Borland C++ 3.1. It has also been successfully compiled under Linux, using G++, the GNU C compiler. Although many of the case studies have also been tested with Turbo C++ 3.1, there appears to be a bug in that compiler that prevents the template set class from compiling correctly.

The Turbo Pascal source code was developed to run on any version of Turbo Pascal from 5.5 onwards (although Delphi users should read the notes on Delphi below). However, it makes little use of OOP extensions.

The Modula-2 source code should be immediately usable on MS-DOS based systems using the shareware compiler marketed by Fitted Software Tools (FST), the Stony Brook Modula-2 compiler marketed by Gogesch Micro Systems, Inc., or the TopSpeed Modula-2 compilers developed by Jensen and Partners International (JPI). Provided the appropriate version of the I/O module `FileIO` is used, it should also compile unchanged on XDS and Stony Brook ISO-compliant compilers, and under Gardens Point Modula-2 on a wide range of systems.

The software in the distribution is currently supplied in the form of compressed, self-extracting MS-DOS executable files. There are eight of these files.

```
cocorc.exe         71 Kb  Minimal Coco/R for C          (Updated November 1999)
cocorm.exe         76 Kb  Minimal Coco/R for Modula-2   (Updated November 1999)
cocorp.exe         65 Kb  Minimal Coco/R for Pascal     (Updated November 1999)

common.exe         93 Kb  Files common to all languages (Updated November 1999)
csources.exe      350 Kb  C++ specific sources          (Updated November 1999)
msources.exe      345 Kb  Modula-2 specific sources     (Updated November 1999)
psources.exe      312 Kb  Pascal specific sources       (Updated November 1999)

fileio.exe        153 Kb  Modula-2 FileIO module        (Updated November 1999)

lha213.exe         43 Kb  LHArc decompressor
readme.1st         11 Kb  Installation and setup instructions
```

A.2 Unpacking the software

To unpack the software, simply follow the steps below. Example MS-DOS commands are shown (these may need alteration, depending on the configuration of your computer). Windows users may

follow an equivalent sequence of operations from within the File Manager or Explorer. The files may also be unpacked directly on other systems using appropriate ports of `LHA`.

- Make a backup copy of the software and keep the original in a safe place.
- Proceed to unpack (a) the language specific case studies (b) the language independent files (c) the language specific compiler generator and, if you are a Modula-2 user, (d) the `FileIO` library.

Unpacking the case study software

- Create a directory to act as the root directory for storing your sources, for example

  ```
  MKDIR C:\SRCES
  ```

  (`C:\SRCES` is used as an example only - please yourself in this regard.)

- Log onto this as the working directory

  ```
  CD C:\SRCES
  ```

- Unpack the chosen source file to the `C:\SRCES` directory. For example, for the C++ sources, execute

  ```
  X:\DOWNLOAD\CSOURCES.EXE
  ```

  where `X:\DOWNLOAD\` is the path to the directory in which the downloaded distribution resides.

- You must also unpack the language independent files to the same directory by executing

  ```
  X:\DOWNLOAD\COMMON.EXE
  ```

  These steps will create a small directory hierarchy under the `C:\SRCES` directory, in which various subdirectories will appear, usually one for each chapter. For example, you will find the source code for the programs in chapter 10 in the directory `C:\SRCES\CHAP10\CPP` (for the C++ versions) or the directory `C:\SRCES\CHAP10\MODULA` (for the Modula-2 versions).

IMPORTANT

- Once you have unpacked the system, read the file `C:\SRCES\README.SRC` for further details on how the directories are laid out.
- You may unpack all three of the language specific sources into the same directory tree if you wish - the C++, Modula-2 and Pascal sources are stored in separate directories under the directory for each chapter.
- If you are using Borland C++ you may need to edit the configuration file `C:\SRCES\COMMON\TURBOC.CFG` to reflect the correct paths and options for your compiler.

Unpacking the compiler generator Coco/R

- Create a directory to act as the root directory for your chosen version of Coco/R, for example

  ```
  MKDIR C:\COCO
  ```

- Log onto this as the working directory

```
CD C:\COCO
```

- Unpack the chosen Coco/R system to the `C:\COCO` directory. For example, for the C++ version, execute

```
X:\DOWNLOAD\COCORC.EXE
```

  This will create a small directory hierarchy under the `C:\COCO\` directory, containing the various components of Coco/R.

IMPORTANT

- Once you have unpacked the system, read the file `C:\COCO\README.1ST` for further details on how the directories are laid out, and how to complete the installation of Coco/R so that it can be executed easily.
- If you wish to install more than one version of Coco/R, please do so into separate directories, for example `C:\COCOC\` for the C++ version, `C:\COCOM\` for the Modula-2 version, `C:\COCOP\` for the Pascal version.
- If you are using Borland C++ you may need to edit the configuration file `C:\SRCES\COMMON\TURBOC.CFG` to reflect the correct paths and options for your compiler.

A.3 The FileIO library (Modula-2 users only)

When this book was first published, standardized I/O for Modula-2 was not yet widely available (and was incompatible with extant Modula-2 compilers). The Modula-2 source code in this distribution attempts to get around this problem by providing (another!) I/O module, called `FileIO`. The definition module for `FileIO` is acceptable to all compilers tested so far; implementations have been supplied for each that differ internally only in a few places.

`FileIO` provides the usual services for opening and closing text files, and for reading and writing strings, words, whole numbers and line marks to such files. It can also handle random access binary files, as block read and write operations are provided. In addition there are some utility procedures, for obtaining command line parameters and environment strings, and for the output of dates and times. The module is of fairly widespread applicability beyond the confines of this text, and is compatible with the modules generated by Coco/R (which assumes the module to be available). As an example of a library module it is really rather too large, but has been developed in this way to minimize the number of non-portable sections and modules needed for implementing the programs in the book.

Modula-2 scanners, parsers, and compilers created by Coco/R assume that you will use the I/O module `FileIO`. If you are a Modula-2 user, you will thus need to install and compile the version of `FileIO` that matches your compiler.

Among the resources you will find a self-extracting file `FILEIO.EXE` that contains the sources of `FileIO` for a variety of Modula-2 compilers.

- Make a directory to contain these sources

```
MKDIR C:\FILEIO
```

- Log onto this as the working directory

```
CD C:\FILEIO
```

- Unpack the `FILEIO.EXE` file to the `C:\FILEIO` directory

```
X:\DOWNLOAD\FILEIO.EXE
```

  This will create a small directory hierarchy under the `C:\FILEIO` directory, in which various subdirectories will appear, one for each compiler.

  In the `C:\FILEIO` directory you will find a definition module `FILEIO.DEF`, and in the subdirectories of `C:\FILEIO` you will find various compiler specific modules, including `FILEIO.MOD`. You will need to proceed as follows, on the assumption that you have a "working" directory `C:\WORK` in which you normally develop programs. (You may, of course, find it preferable to install `FileIO` in the library directory or directories for your Modula-2 compiler.)

```
CD C:\WORK
COPY C:\FILEIO\FILEIO.DEF
COPY C:\FILEIO\xxx

where xxx =
    JPI   (TopSpeed compilers)
    FST   (Fitted Systems Tools compilers)
    LOG   (Logitech compilers)
    STO   (StonyBrook compilers)
    GPMPC (Gardens Point PC compiler)
    etc
```

  Follow this by compiling `FILEIO.DEF` and `FILEIO.MOD`.

The sources supplied will act as models of implementations for compilers not mentioned above. In case of difficulty, please contact the author.

A.4 Setting up the software for a particular case study

Installation of many (but not all) of the case studies requires you simply to

- Log onto to the appropriate source drive and subdirectory, for example

```
C:
CD C:\SRCES\CHAP15\CPP
```

- Issue the command

```
SETUP
```

  with no parameters. This will indicate what parameters might be added to the command to provide a variant appropriate for the case studies of that chapter.

- For example, follow this by obeying a `SETUP` command like

```
SETUP C:\WORK 5
```

  where `C:\WORK` is a directory that will be created if necessary, and into which the appropriate

components for SETUP option 5 will be copied.

- Log onto the working directory

```
CD C:\WORK
```

- Issue the MAKE command (C++), or equivalent Coco/R and compiler commands (Pascal or Modula-2) to compile the selected case study.

IMPORTANT

- The "makefiles" supplied are intended for use with Borland C++ release 3.1; they may need slight alteration. The configuration file TURBOC.CFG may also need alteration.
- It is crucial that you are logged onto an appropriate directory before you issue the SETUP command. If this is not the case, you might execute some other SETUP command and become confused!
- Some of the simpler case studies - typically those consisting of only one source file - are not managed by a SETUP process. Consult the file C:\SRCES\README.SRC for further details.

The Turbo Pascal case studies, and the output from Coco/R for Pascal will not compile directly under Delphi. They require WinCrt to be added to the list of units "used" (edit the compiler frame files for Coco/R). Furthermore those systems (like the assemblers and compilers) that provide for interactive input/output in their interpreters require use of WinCrt.AssignCrt instead of the current calls to Assign with an empty filename (for example in STKMC.PAS and MC.PAS).

A.5 Coco/R distributions

The compiler generator Coco/R used in this book was originally developed in Oberon by Hanspeter Mössenböck, who also did a port to Modula-2 for the Apple MacMeth system. A further port was done to TopSpeed Modula-2 by Marc Brandis and Christof Brass. This was refined and extended by the author in conjunction with John Gough and Hanspeter Mössenböck, to the point where a single version runs on most Modula-2 compilers available under MS-DOS, as well as the Mocka and Gardens Point compilers available for Unix (and other) systems, including Linux and Free BSD.

A port of Coco/R to Turbo Pascal was done by the author in conjunction with Volker Pohlers.

Coco/R was ported to C by Francisco Arzu, yielding a version that can generate either C or C++ compilers.

Coco/R has also been also ported to Java by Hanspeter Mössenböck.

For details of how to obtain the latest versions of the complete Coco/R distributions for a variety of languages and operating systems visit the Coco/R home page at
http://cs.ru.ac.za/homes/cspt/cocor.htm.

The original report on Coco/R (Mössenböck, 1990a) can be obtained from

ftp://ftp.ssw.uni-linz.ac.at/pub/Papers/Coco.Report.ps.Z
ftp://cs.ru.ac.za/pub/coco/Coco.Report.ps.Z

Professor Mössenböck may be contacted at the address below

Prof. Hanspeter Mössenböck
Institute of Computer Science
University of Linz
Alternbergerstr 69,
A-4040 Linz, Austria
Tel: +43-732-2468-9700
e-mail: moessenboeck@ssw.uni-linz.ac.at

A.6 Other compiler tools, free compilers and similar resources

The subject of compiler writing and the development of compilers is one of the classical fields of Computer Science for which there are now many freely available resources. The interested reader might do worse than to explore some of the following leads:

FAQ's, topics and archives from the Comp.Compilers news group
http://www.iecc.com/compilers

The German National Resource Centre catalogue of compiler construction tools
http://www.first.gmd.de/cogent/catalog

Idiom Consulting's catalogue of free compilers
http://www.idiom.com/free-compilers

Compiler Connection
http://www.compilerconnection.com

Burks (Brighton University Resource Kit for Students)
http://burks.bton.ac.uk/burks

Dragon Fodder
http://www.km-cd.com/dragon_fodder

Jack Crenshaw's "Let's build a compiler"
http://www.iecc.com/compilers/crenshaw/

The PCCTS compiler construction kit mentioned in Chapter 10 is available from

http://www.ANTLR.org/pccts133.html

Freely available early versions of the Cocktail compiler construction tools mentioned in Chapter 10 may be obtained by anonymous ftp from

ftp://ftp.ira.uka.de/pub/programming/cocktail
ftp://144ftp.info.uni-karlsruhe.de/pub/cocktail

For the commercial version and support visit http://www.first.gmd.de/cocktail/ or contact Josef Grosch.

`mtc`, the Modula-2 to C translator program mentioned in Chapter 2 is available by anonymous ftp

from

ftp://ftp.psg.com:/pub/modula-2/grosch/mtc.tar.Z
ftp://ftp.gmd.de/gmd/cocktail/mtc.tar.Z

`p2c`, the Pascal to C translator program mentioned in Chapter 2, and `cperf`, the perfect hash function generator mentioned in Chapter 14, are available by anonymous ftp from any of the sites that mirror the Free Software Foundation GNU archives. The primary server for these archives is at ftp://prep.ai.mit.edu. Among many others, the Linux sites, such as those at ftp://tsx-11.mit.edu or ftp://sunsite.unc.edu and ftp://src.doc.ic.ac.uk also carry copies of the GNU archives.

Also to be found among these archives is `gcc`, the well-known GNU C compiler, which can also be found at ftp://gcc.gnu.org or at one of many mirror sites

As another example of a freely available C compiler, `lcc` is a retargetable compiler for ANSI C described in "A Retargetable C Compiler: Design and Implementation" by Fraser, Hanson and Hansen (Benjamin/Cummings, 1995). `lcc` is available, along with documentation and a sample chapter of the book at

http://www.CS.Princeton.EDU/software/lcc/

Versions of the Gardens Point Modula-2 compiler for DOS, Linux and FreeBSD are available from

ftp://ftp.fit.qut.edu.au/pub/gpm_modula2
ftp://ftp.psg.com/pub/modula-2/gpm

Versions of the Mocka Modula-2 compiler for Linux and FreeBSD are available from

http://www.first.gmd.de/mocka/

The shareware FST Modula-2 compiler for MS-DOS systems is available by anonymous ftp from

ftp://ftp.psg.com/pub/modula-2/fst/fst-40s.lzh
ftp://cs.ru.ac.za/pub/languages/fst-40s.lzh

The self-extracting files in this distribution were compressed and packed using the freely available program `LHA.EXE` developed by Haruyasu Yoshizaki. In terms of the distribution agreement for this program, the complete package for LHA.EXE is itself supplied as a self-extracting executable, LHA213.EXE. You are quite welcome to unpack this file, although it is not needed for the operations described above.

Further to comply with the distribution agreement, the copyright notice for this package is given below

```
4. Our distribution Policy

   This software, this document and LHA.EXE, is a copyright-reserved free
   program.  You may use, copy and distribute this software free of charge
   under the following conditions.

1. Never change Copyright statement.

2. The enclosed documents must be distributed with as a package.
```

3. When you have changed the program, or implemented the program for other OS or environment, then you must specify the part you have changed. Also make a clear statement as to your name and MAIL address or phone number.

4. The author is not liable for any damage on your side caused by the use of this program.

5. The author has no duty to remedy for the deficiencies of the program.

6. When you are to distribute this software with publications or with your product, you have to print the copyright statement somewhere on the disk or on the package. You cannot distribute this software with copyprotected products.

Disclaimer

While every attempt has been made to ensure that the software in this distribution performs properly, the author can accept no liability for any damage or loss, including special, incidental, or consequential, caused by the use of the software, directly or indirectly.

However, please bring any problems that you may experience to the attention of the author:

Pat Terry
Computer Science Department
Rhodes University
GRAHAMSTOWN 6140
South Africa
Tel: +27-46-622-8292
FAX: +27-46-636-1915
e-mail: cspt@cs.ru.ac.za
e-mail: p.terry@ru.ac.za

Trademarks

Products and services that are referred to in this book may be either trademarks and/or registered trademarks of their respective owners. The author makes no claim to these trademarks.

Borland C++, Turbo C++, Delphi and Turbo Pascal are trademarks of Borland International Corporation.

GNU C Compiler is a trademark of the Free Software Foundation.

IBM and IBM PC are trademarks of International Business Machines Corporation.

Microsoft, MS and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation.

Stony Brook Software and QuickMod are trademarks of Gogesch Micro Systems, Inc.

TopSpeed is a registered trademark of Jensen and Partners, International.

Any other trademarks inadvertently used here are also duly acknowledged.

Appendix B

Source code for the Clang compiler/interpreter

This appendix gives the complete source code for a hand-crafted compiler for the Clang language as developed by the end of Chapter 18.

cln.cpp | misc.h | set.h | srce.h | srce.cpp | report.h | report.cpp | scan.h | scan.cpp | parser.h | parser.cpp | table.h | table.cpp | cgen.h | cgen.cpp | stkmc.h | stkmc.cpp

```
----- cln.cpp ------------------------------------------------------------------

// Clang Compiler/Interpreter
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "srce.h"
#include "scan.h"
#include "parser.h"
#include "table.h"
#include "report.h"
#include "stkmc.h"
#include "cgen.h"

#define usage "USAGE: CLN source [listing]\n"

static char SourceName[256], ListName[256], CodeName[256];

TABLE  *Table;
CGEN   *CGen;
STKMC  *Machine;
REPORT *Report;

class clangReport : public REPORT {
  public:
    clangReport(SRCE *S)   { Srce = S; }
    virtual void error(int errorcode)
      { Srce->reporterror(errorcode); errors = true; }
  private:
    SRCE *Srce;
};

class clangSource : public SRCE {
  public:
    clangSource(char *sname, char *lname, char *ver, bool lw)
      : SRCE(sname, lname, ver, lw) {};
    virtual void startnewline() { fprintf(lst, "%4d : ", CGen->gettop()); }
};

void main(int argc, char *argv[])
{ char reply;
  int codelength, initsp;

  // check on correct parameter usage
  if (argc == 1) { printf(usage); exit(1); }
  strcpy(SourceName, argv[1]);
  if (argc > 2) strcpy(ListName, argv[2]);
  else appendextension(SourceName, ".lst", ListName);

  clangSource *Source = new clangSource(SourceName, ListName, STKMC_version, true);
  Report         = new clangReport(Source);
  CGen           = new CGEN(Report);
  SCAN *Scanner  = new SCAN(Source, Report);
  Table          = new TABLE(Report);
  PARSER *Parser = new PARSER(CGen, Scanner, Table, Source, Report);
  Machine        = new STKMC();

  Parser->parse();
  CGen->getsize(codelength, initsp);

  appendextension(SourceName, ".cod", CodeName);
```

```
  Machine->listcode(CodeName, codelength);

  if (Report->anyerrors())
    printf("Compilation failed\n");
  else
  { printf("Compilation successful\n");
    while (true)
    { printf("\nInterpret? (y/n) ");
      do
      { scanf("%c", &reply);
      } while (toupper(reply) != 'N' && toupper(reply) != 'Y');
      if (toupper(reply) == 'N') break;
      scanf("%*[^\n]"); getchar();
      Machine->interpret(codelength, initsp);
    }
  }
  delete Source;
  delete Scanner;
  delete Parser;
  delete Table;
  delete Report;
  delete CGen;
  delete Machine;
}


----- misc.h ----------------------------------------------------------------

// Various common items

#ifndef MISC_H
#define MISC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define  boolean  int
#define  bool     int
#define  true     1
#define  false    0
#define  TRUE     1
#define  FALSE    0
#define  maxint   INT_MAX

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  define  pathsep '\\'
#else
#  define  pathsep '/'
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{ int i;
  char old[256];
  strcpy(old, oldstr);
  i = strlen(old);
  while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
  if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
  if (ext[0] == '.') sprintf(newstr,"%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

#endif /* MISC_H */


----- set.h -----------------------------------------------------------------

// simple set operations
// Thanks to E.P. Wentworth for useful ideas!

#ifndef SET_H
#define SET_H

template <int maxElem>
class Set {                         // { 0 .. maxElem }
  public:
    Set()                           // Construct { }
    { clear(); }
```

```
    Set(int e1)                     // Construct { e1 }
    { clear(); incl(e1); }

    Set(int e1, int e2)             // Construct { e1, e2 }
    { clear(); incl(e1); incl(e2); }

    Set(int e1, int e2, int e3)     // Construct { e1, e2, e3 }
    { clear(); incl(e1); incl(e2); incl(e3); }

    Set(int n, int e1[])            // Construct { e[0] .. e[n-1] }
    { clear(); for (int i = 0; i < n; i++) incl(e1[i]); }

    void incl(int e)                // Include e
    { if (e >= 0 && e <= maxElem) bits[wrd(e)] |= bitmask(e); }

    void excl(int e)                // Exclude e
    { if (e >= 0 && e <= maxElem) bits[wrd(e)] &= ~bitmask(e); }

    int memb(int e)                 // Test membership for e
    { if (e >= 0 && e <= maxElem) return((bits[wrd(e)] & bitmask(e)) != 0);
      else return 0;
    }

    int isempty(void)               // Test for empty set
    { for (int i = 0; i < length; i++) if (bits[i]) return 0;
      return 1;
    }

    Set operator + (const Set &s)   // Union with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] | s.bits[i];
      return r;
    }

    Set operator * (const Set &s)   // Intersection with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] & s.bits[i];
      return r;
    }

    Set operator - (const Set &s)   // Difference with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] & ~s.bits[i];
      return r;
    }

    Set operator / (const Set &s)   // Symmetric difference with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] ^ s.bits[i];
      return r;
    }

  private:
    unsigned char       bits[(maxElem + 8) / 8];
    int                 length;
    int wrd(int i)      { return(i / 8); }
    int bitmask(int i)  { return(1 << (i % 8)); }
    void clear()        { length = (maxElem + 8) / 8;
                          for (int i = 0; i < length; i++) bits[i] = 0;
                        }
};

#endif /* SET_H */



----- srce.h ----------------------------------------------------------------

// Source handler for various parsers/compilers
// P.D. Terry, Rhodes University, 1996

#ifndef SRCE_H
#define SRCE_H

#include "misc.h"

const int linemax = 129;        // limit on source line length

class SRCE {
  public:
    FILE *lst;                  // listing file
    char ch;                    // latest character read
```

```
    void nextch(void);
    // Returns ch as the next character on this source line, reading a new
    // line where necessary.  ch is returned as NUL if src is exhausted.

    bool endline(void)       { return (charpos == linelength); }
    // Returns true when end of current line has been reached

    void listingon(void)         { listing = true; }
    // Requests source to be listed as it is read

    void listingoff(void)        { listing = false; }
    // Requests source not to be listed as it is read

    void reporterror(int errorcode);
    // Points out error identified by errorcode with suitable message

    virtual void startnewline()  {;}
    // called at start of each line

    int getline(void)            { return linenumber; }
    // returns current line number

    SRCE(char *sourcename, char *listname, char *version, bool listwanted);
    // Open src and lst files using given names.
    // Resets internal state in readiness for starting to scan.
    // Notes whether listwanted.  Displays version information on lst file.

    ~SRCE();
    // close src and lst files

  private:
    FILE *src;                  // Source file
    int linenumber;             // Current line number
    int charpos;                // Character pointer
    int minerrpos;              // Last error position
    int linelength;             // Line length
    char line[linemax + 1];     // Last line read
    bool listing;               // true if listing required
};

#endif /*SRCE_H*/


----- srce.cpp ------------------------------------------------------------

// Source handler for various parsers/compilers
// P.D. Terry, Rhodes University, 1996

#include "srce.h"

void SRCE::nextch(void)
{ if (ch == '\0') return;        // input exhausted
  if (charpos == linelength)     // new line needed
  { linelength = 0; charpos = 0; minerrpos = 0; linenumber++;
    startnewline();
    do
    { ch = getc(src);
      if (ch != '\n' && !feof(src))
      { if (listing) putc(ch, lst);
        if (linelength < linemax) { line[linelength] = ch; linelength++; }
      }
    } while (!(ch == '\n' || feof(src)));
    if (listing) putc('\n', lst);
    if (feof(src))
      line[linelength] = '\0';   // mark end with a nul character
    else
      line[linelength] = ' ';    // mark end with an explicit space
    linelength++;
  }
  ch = line[charpos]; charpos++; // pass back unique character
}

// reporterror has been coded like this (rather than use a static array of
// strings initialized by the array declarator) to allow for easy extension
// in exercises

void SRCE::reporterror(int errorcode)
{ if (charpos > minerrpos) // suppress cascading messages
  { startnewline(); fprintf(lst, "%*c", charpos, '^');
    switch (errorcode)
    { case 1:   fprintf(lst, "Incomplete string\n"); break;
      case 2:   fprintf(lst, "; expected\n"); break;
      case 3:   fprintf(lst, "Invalid start to block\n"); break;
```

```
      case 4:   fprintf(lst, "Invalid declaration sequence\n"); break;
      case 5:   fprintf(lst, "Invalid procedure header\n"); break;
      case 6:   fprintf(lst, "Identifier expected\n"); break;
      case 7:   fprintf(lst, ":= in wrong context\n"); break;
      case 8:   fprintf(lst, "Number expected\n"); break;
      case 9:   fprintf(lst, "= expected\n"); break;
      case 10:  fprintf(lst, "] expected\n"); break;
      case 13:  fprintf(lst, ", or ) expected\n"); break;
      case 14:  fprintf(lst, "Invalid factor\n"); break;
      case 15:  fprintf(lst, "Invalid start to statement\n"); break;
      case 17:  fprintf(lst, ") expected\n"); break;
      case 18:  fprintf(lst, "( expected\n"); break;
      case 19:  fprintf(lst, "Relational operator expected\n"); break;
      case 20:  fprintf(lst, "Operator expected\n"); break;
      case 21:  fprintf(lst, ":= expected\n"); break;
      case 23:  fprintf(lst, "THEN expected\n"); break;
      case 24:  fprintf(lst, "END expected\n"); break;
      case 25:  fprintf(lst, "DO expected\n"); break;
      case 31:  fprintf(lst, ", or ; expected\n"); break;
      case 32:  fprintf(lst, "Invalid symbol after a statement\n"); break;
      case 34:  fprintf(lst, "BEGIN expected\n"); break;
      case 35:  fprintf(lst, "Invalid symbol after block\n"); break;
      case 36:  fprintf(lst, "PROGRAM expected\n"); break;
      case 37:  fprintf(lst, ". expected\n"); break;
      case 38:  fprintf(lst, "COEND expected\n"); break;
      case 200: fprintf(lst, "Constant out of range\n"); break;
      case 201: fprintf(lst, "Identifier redeclared\n"); break;
      case 202: fprintf(lst, "Undeclared identifier\n"); break;
      case 203: fprintf(lst, "Unexpected parameters\n"); break;
      case 204: fprintf(lst, "Unexpected subscript\n"); break;
      case 205: fprintf(lst, "Subscript required\n"); break;
      case 206: fprintf(lst, "Invalid class of identifier\n"); break;
      case 207: fprintf(lst, "Variable expected\n"); break;
      case 208: fprintf(lst, "Too many formal parameters\n"); break;
      case 209: fprintf(lst, "Wrong number of parameters\n"); break;
      case 210: fprintf(lst, "Invalid assignment\n"); break;
      case 211: fprintf(lst, "Cannot read this type of variable\n"); break;
      case 212: fprintf(lst, "Program too long\n"); break;
      case 213: fprintf(lst, "Too deeply nested\n"); break;
      case 214: fprintf(lst, "Invalid parameter\n"); break;
      case 215: fprintf(lst, "COBEGIN only allowed in main program\n"); break;
      case 216: fprintf(lst, "Too many concurrent processes\n"); break;
      case 217: fprintf(lst, "Only global procedure calls allowed here\n"); break;
      case 218: fprintf(lst, "Type mismatch\n"); break;
      case 219: fprintf(lst, "Unexpected expression\n"); break;
      case 220: fprintf(lst, "Missing expression\n"); break;
      case 221: fprintf(lst, "Boolean expression required\n"); break;
      case 222: fprintf(lst, "Invalid expression\n"); break;
      case 223: fprintf(lst, "Index out of range\n"); break;
      case 224: fprintf(lst, "Division by zero\n"); break;
      default:
        fprintf(lst, "Compiler error\n"); printf("Compiler error\n");
        if (lst != stdout) fclose(lst); exit(1);
    }
  }
  minerrpos = charpos + 1;
}

SRCE::~SRCE()
{ if (src != NULL)   { fclose(src); src = NULL; }
  if (lst != stdout) { fclose(lst); lst = NULL; }
}

SRCE::SRCE(char *sourcename, char *listname, char *version, bool listwanted)
{ src = fopen(sourcename, "r");
  if (src == NULL) { printf("Could not open input file\n"); exit(1); }
  lst = fopen(listname, "w");
  if (lst == NULL) { printf("Could not open listing file\n"); lst = stdout; }
  listing = listwanted;
  if (listing) fprintf(lst, "%s\n\n", version);
  charpos = 0; linelength = 0; linenumber = 0; ch = ' ';
}


----- report.h ----------------------------------------------------------------

// Handle reporting of errors when parsing or compiling Clang programs
// P.D. Terry, Rhodes University, 1996

#ifndef REPORT_H
#define REPORT_H

#include "misc.h"
```

```
class REPORT {
  public:
    REPORT ()               { errors = false; }
    // Initialize error reporter

    virtual void error(int errorcode);
    // Reports on error designated by suitable errorcode number

    bool anyerrors(void)  { return errors; }
    // Returns true if any errors have been reported

  protected:
    bool errors;
};

#endif /*REPORT_H*/


----- report.cpp ----------------------------------------------------------

// Handle reporting of errors when parsing or compiling Clang programs
// P.D. Terry, Rhodes University, 1996

#include "report.h"

void REPORT::error(int errorcode)
{ printf("Error %d\n", errorcode); errors = true; exit(1); }


------ scan.h -------------------------------------------------------------

// Lexical analyzer for Clang parsers/compilers
// P.D. Terry, Rhodes University, 1996

#ifndef SCAN_H
#define SCAN_H

#include "misc.h"
#include "report.h"
#include "srce.h"

const int lexlength = 128;
typedef char lexeme[lexlength + 1];

const int alfalength = 10;
typedef char alfa[alfalength + 1];

enum SCAN_symtypes {
  SCAN_unknown, SCAN_becomes, SCAN_lbracket, SCAN_times, SCAN_slash, SCAN_plus,
  SCAN_minus, SCAN_eqlsym, SCAN_neqsym, SCAN_lsssym, SCAN_leqsym, SCAN_gtrsym,
  SCAN_geqsym, SCAN_thensym, SCAN_dosym, SCAN_rbracket, SCAN_rparen, SCAN_comma,
  SCAN_lparen, SCAN_number, SCAN_stringsym, SCAN_identifier, SCAN_coendsym,
  SCAN_endsym, SCAN_ifsym, SCAN_whilesym, SCAN_stacksym, SCAN_readsym,
  SCAN_writesym, SCAN_returnsym, SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym,
  SCAN_semicolon, SCAN_beginsym, SCAN_constsym, SCAN_varsym, SCAN_procsym,
  SCAN_funcsym, SCAN_period, SCAN_progsym, SCAN_eofsym
};

struct SCAN_symbols {
  SCAN_symtypes sym;    // symbol type
  int num;              // value
  lexeme name;          // lexeme
};

class SCAN {
  public:
    void getsym(SCAN_symbols &SYM);
    // Obtains the next symbol in the source text

    SCAN(SRCE *S, REPORT *R);
    // Initializes scanner

  protected:
    REPORT *Report;               // Associated error reporter
    SRCE *Srce;                   // Associated source handler
    static struct keyword {
      alfa resword;
      SCAN_symtypes ressym;
    } table[];                    // Look up table words/symbols
    int keywords;                 // Actual number of them
    SCAN_symtypes singlesym[256]; // One character symbols
};
```

```
#endif /*SCAN_H*/


----- scan.cpp ------------------------------------------------------------

// Lexical analyzer for Clang parsers/compilers
// P.D. Terry, Rhodes University, 1996

#include "scan.h"

SCAN::keyword SCAN::table[] = {  // this must be in alphabetic order
  { "BEGIN",     SCAN_beginsym },
  { "COBEGIN",   SCAN_cobegsym },
  { "COEND",     SCAN_coendsym },
  { "CONST",     SCAN_constsym },
  { "DO",        SCAN_dosym },
  { "END",       SCAN_endsym },
  { "FUNCTION",  SCAN_funcsym },
  { "IF",        SCAN_ifsym },
  { "PROCEDURE", SCAN_procsym },
  { "PROGRAM",   SCAN_progsym },
  { "READ",      SCAN_readsym },
  { "RETURN",    SCAN_returnsym },
  { "SIGNAL",    SCAN_signalsym },
  { "STACKDUMP", SCAN_stacksym },
  { "THEN",      SCAN_thensym },
  { "VAR",       SCAN_varsym },
  { "WAIT",      SCAN_waitsym },
  { "WHILE",     SCAN_whilesym },
  { "WRITE",     SCAN_writesym },
};

void SCAN::getsym(SCAN_symbols &SYM)
{ int length;       // index into SYM.Name
  int digit;        // value of digit character
  int l, r, look;   // for binary search
  bool overflow;    // in numeric conversion
  bool endstring;   // in string analysis

  while (Srce->ch == ' ') Srce->nextch();  // Ignore spaces between tokens
  SYM.name[0] = Srce->ch; SYM.name[1] = '\0'; SYM.num = 0; length = 0;
  SYM.sym = singlesym[Srce->ch];   // Initial assumption
  if (isalpha(Srce->ch))           // Identifier or reserved word
  { while (isalnum(Srce->ch))
    { if (length < lexlength) { SYM.name[length] = toupper(Srce->ch); length++; }
      Srce->nextch();
    }
    SYM.name[length] = '\0';       // Terminate string properly
    l = 0; r = keywords - 1;
    do                             // Binary search
    { look = (l + r) / 2;
      if (strcmp(SYM.name, table[look].resword) <= 0) r = look - 1;
      if (strcmp(SYM.name, table[look].resword) >= 0) l = look + 1;
    } while (l <= r);
    if (l - 1 > r)
      SYM.sym = table[look].ressym;
    else
      SYM.sym = SCAN_identifier;
  }
  else if (isdigit(Srce->ch))      // Numeric literal
  { SYM.sym = SCAN_number;
    overflow = false;
    while (isdigit(Srce->ch))
    { digit = Srce->ch - '0';
      // Check imminent overflow
      if (SYM.num <= (maxint - digit) / 10)
        SYM.num = SYM.num * 10 + digit;
      else
        overflow = true;
      if (length < lexlength) { SYM.name[length] = toupper(Srce->ch); length++; }
      Srce->nextch();
    }
    if (overflow) Report->error(200);
    SYM.name[length] = '\0';       // Terminate string properly
  }
  else switch (Srce->ch)
  { case ':':
      Srce->nextch();
      if (Srce->ch == '=')
      { SYM.sym = SCAN_becomes; strcpy(SYM.name, ":="); Srce->nextch(); }
      // else SYM.sym := SCAN_unknown; SYM.name := ":"
      break;
```

```
    case '<':
      Srce->nextch();
      if (Srce->ch == '=')
        { SYM.sym = SCAN_leqsym; strcpy(SYM.name, "<="); Srce->nextch(); }
      else if (Srce->ch == '>')
        { SYM.sym = SCAN_neqsym; strcpy(SYM.name, "<>"); Srce->nextch(); }
      // else SYM.sym := SCAN_lsssym; SYM.name := "<"
      break;

    case '>':
      Srce->nextch();
      if (Srce->ch == '=')
      { SYM.sym = SCAN_geqsym; strcpy(SYM.name, ">="); Srce->nextch(); }
      // else SYM.sym := SCAN_gtrsym; SYM.name := ">"
      break;

    case '\'':   // String literal
      Srce->nextch();
      SYM.sym = SCAN_stringsym;
      endstring = false;
      do
      { if (Srce->ch == '\'')
        { Srce->nextch(); endstring = (Srce->ch != '\''); }
        if (!endstring)
        { if (length < lexlength) { SYM.name[length] = Srce->ch; length++; }
          Srce->nextch();
        }
      } while (!(endstring || Srce->endline()));
      if (!endstring) Report->error(1);
      SYM.name[length] = '\0';     // Terminate string properly
      break;

    case '\0':
      SYM.sym = SCAN_eofsym; break;

    default:
      // implementation defined symbols - SYM.sym := singlesym[Srce->ch]
      Srce->nextch(); break;
  }
}

SCAN::SCAN(SRCE *S, REPORT *R)
{ Srce = S; Report = R;
  // Define one char symbols
  for (int i = 0; i <= 255; i++) singlesym[i] = SCAN_unknown;
  singlesym['+'] = SCAN_plus;      singlesym['-'] = SCAN_minus;
  singlesym['*'] = SCAN_times;     singlesym['/'] = SCAN_slash;
  singlesym['('] = SCAN_lparen;    singlesym[')'] = SCAN_rparen;
  singlesym['['] = SCAN_lbracket;  singlesym[']'] = SCAN_rbracket;
  singlesym['='] = SCAN_eqlsym;    singlesym[';'] = SCAN_semicolon;
  singlesym[','] = SCAN_comma;     singlesym['.'] = SCAN_period;
  singlesym['<'] = SCAN_lsssym;    singlesym['>'] = SCAN_gtrsym;
  keywords = sizeof(table) / sizeof(keyword);
  Srce->nextch();
}


----- parser.h ----------------------------------------------------------------

// Parser for Clang source
// P.D. Terry, Rhodes University, 1996

#ifndef PARSER_H
#define PARSER_H

#include "scan.h"
#include "report.h"
#include "table.h"
#include "srce.h"
#include "set.h"
#include "cgen.h"

typedef Set<SCAN_eofsym> symset;

class PARSER {
  public:
    PARSER(CGEN *C, SCAN *L, TABLE *T, SRCE *S, REPORT *R);
    // Initializes parser

    void parse(void);
    // Parses and compiles the source code
```

```
  private:
    symset RelOpSyms, FirstDeclaration, FirstBlock, FirstFactor,
           FirstExpression, FirstStatement, EmptySet;
    SCAN_symbols SYM;
    REPORT *Report;
    SCAN *Scanner;
    TABLE *Table;
    SRCE *Srce;
    CGEN *CGen;
    bool debugging;
    int blocklevel;
    TABLE_idclasses blockclass;
    void GetSym(void);
    void accept(SCAN_symtypes expected, int errorcode);
    void test(symset allowed, symset beacons, int errorcode);
    CGEN_operators op(SCAN_symtypes s);
    void OneConst(void);
    void ConstDeclarations(void);
    void OneVar(int &framesize);
    void VarDeclarations(int &framesize);
    void OneFormal(TABLE_entries &procentry, TABLE_index &parindex);
    void FormalParameters(TABLE_entries &procentry);
    void ProcDeclaration(symset followers);
    void Designator(TABLE_entries entry, symset followers, int errorcode);
    void ReferenceParameter(void);
    void OneActual(symset followers, TABLE_entries procentry, int &actual);
    void ActualParameters(TABLE_entries procentry, symset followers);
    void Variable(symset followers, int errorcode);
    void Factor(symset followers);
    void Term(symset followers);
    void Expression(symset followers);
    void Condition(symset followers);
    void CompoundStatement(symset followers);
    void Assignment(symset followers, TABLE_entries entry);
    void ProcedureCall(symset followers, TABLE_entries entry);
    void IfStatement(symset followers);
    void WhileStatement(symset followers);
    void ReturnStatement(symset followers);
    void WriteElement(symset followers);
    void WriteStatement(symset followers);
    void ReadStatement(symset followers);
    void ProcessCall(symset followers, int &processes);
    void CobeginStatement(symset followers);
    void SemaphoreStatement(symset followers);
    void Statement(symset followers);
    void Block(symset followers, int blklevel, TABLE_idclasses blkclass,
               int initialframesize);
    void ClangProgram(void);
};

#endif /*PARSER_H*/


----- parser.cpp ----------------------------------------------------------

// Parser for Clang level 4 - incorporates error recovery and code generation
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "parser.h"
#include "report.h"
#include "table.h"
#include "scan.h"
#include "srce.h"
#include "set.h"

static int relopsyms [] = {SCAN_eqlsym, SCAN_neqsym, SCAN_gtrsym,
                           SCAN_geqsym, SCAN_lsssym, SCAN_leqsym};

static int firstDeclaration [] = {SCAN_constsym, SCAN_varsym, SCAN_procsym,
                                  SCAN_funcsym};

static int firstBlock [] = {SCAN_constsym, SCAN_varsym, SCAN_procsym,
                            SCAN_funcsym, SCAN_beginsym};

static int firstFactor [] = {SCAN_identifier, SCAN_number, SCAN_lparen,
                             SCAN_stringsym};

static int firstExpression [] = {SCAN_identifier, SCAN_number, SCAN_lparen,
                                 SCAN_plus, SCAN_minus};
```

```
static int firstStatement [] = {SCAN_identifier, SCAN_beginsym, SCAN_ifsym,
                                SCAN_whilesym, SCAN_returnsym,
                                SCAN_writesym, SCAN_readsym, SCAN_stacksym,
                                SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym};

PARSER::PARSER(CGEN *C, SCAN *L, TABLE *T, SRCE *S, REPORT *R) :
  RelOpSyms(sizeof(relopsyms)/sizeof(int), relopsyms),
  FirstDeclaration(sizeof(firstDeclaration)/sizeof(int), firstDeclaration),
  FirstBlock(sizeof(firstBlock)/sizeof(int), firstBlock),
  FirstFactor(sizeof(firstFactor)/sizeof(int), firstFactor),
  FirstExpression(sizeof(firstExpression)/sizeof(int), firstExpression),
  FirstStatement(sizeof(firstStatement)/sizeof(int), firstStatement),
  EmptySet()
{ CGen = C, Scanner = L; Report = R; Table = T; Srce = S; }

void PARSER::GetSym(void)
{ Scanner->getsym(SYM); }

void PARSER::accept(SCAN_symtypes expected, int errorcode)
{ if (SYM.sym == expected) GetSym(); else Report->error(errorcode); }

void PARSER::test(symset allowed, symset beacons, int errorcode)
// Test whether current Sym is Allowed, and recover if not
{ if (allowed.memb(SYM.sym)) return;
  Report->error(errorcode);
  symset stopset = allowed + beacons;
  while (!stopset.memb(SYM.sym)) GetSym();
}

CGEN_operators PARSER::op(SCAN_symtypes s)
// Map symbol to corresponding code operator
{ switch (s)
  { case SCAN_plus:   return CGEN_opadd;
    case SCAN_minus:  return CGEN_opsub;
    case SCAN_times:  return CGEN_opmul;
    case SCAN_slash:  return CGEN_opdvd;
    case SCAN_eqlsym: return CGEN_opeql;
    case SCAN_neqsym: return CGEN_opneq;
    case SCAN_gtrsym: return CGEN_opgtr;
    case SCAN_geqsym: return CGEN_opgeq;
    case SCAN_lsssym: return CGEN_oplss;
    case SCAN_leqsym: return CGEN_opleq;
  }
}

// ++++++++++++++++++++++++ Declaration Part ++++++++++++++++++++++++++

void PARSER::OneConst(void)
// OneConst = ConstIdentifier "=" Number ";"  .
{ TABLE_entries constentry;
  TABLE_index constindex;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  sprintf(constentry.name, "%.*s", TABLE_alfalength, SYM.name);
  constentry.idclass = TABLE_consts;
  GetSym();
  if (SYM.sym == SCAN_becomes || SYM.sym == SCAN_eqlsym)
  { if (SYM.sym == SCAN_becomes) Report->error(7);
    GetSym();
    if (SYM.sym != SCAN_number)
      { constentry.c.value = 0; Report->error(8); }
    else
      { constentry.c.value = SYM.num; GetSym(); }
  }
  else Report->error(9);
  Table->enter(constentry, constindex);
  accept(SCAN_semicolon, 2);
}

void PARSER::ConstDeclarations(void)
// ConstDeclarations = "CONST" OneConst { OneConst } .
{ GetSym();
  OneConst();
  while (SYM.sym == SCAN_identifier) OneConst();
}

void PARSER::OneVar(int &framesize)
// OneVar = VarIdentifier [ UpperBound ] .
// UpperBound = "[" Number "]" .
{ TABLE_entries varentry;
  TABLE_index varindex;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  varentry.idclass = TABLE_vars;
  varentry.v.size = 1;
```

```
  varentry.v.scalar = true;
  varentry.v.ref = false;
  varentry.v.offset = framesize + 1;
  sprintf(varentry.name, "%.*s", TABLE_alfalength, SYM.name);
  GetSym();
  if (SYM.sym == SCAN_lbracket)
  { // array declaration
    GetSym();
    varentry.v.scalar = false;
    if (SYM.sym == SCAN_identifier || SYM.sym == SCAN_number)
    { if (SYM.sym == SCAN_identifier)
        Report->error(8);
      else
        varentry.v.size = SYM.num + 1;
      GetSym();
    }
    else
      Report->error(8);
    accept(SCAN_rbracket, 10);
  }
  Table->enter(varentry, varindex);
  framesize += varentry.v.size;
}

void PARSER::VarDeclarations(int &framesize)
// VarDeclarations = "VAR" OneVar { "," OneVar } ";" .
{ GetSym();
  OneVar(framesize);
  while (SYM.sym == SCAN_comma || SYM.sym == SCAN_identifier)
    { accept(SCAN_comma, 31); OneVar(framesize); }
  accept(SCAN_semicolon, 2);
}

void PARSER::OneFormal(TABLE_entries &procentry, TABLE_index &parindex)
// OneFormal := ParIdentifier [ "[" "]" ] .
{ TABLE_entries parentry;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  parentry.idclass = TABLE_vars;
  parentry.v.size = 1;
  parentry.v.scalar = true;
  parentry.v.ref = false;
  parentry.v.offset = procentry.p.paramsize + CGEN_headersize + 1;
  sprintf(parentry.name, "%.*s", TABLE_alfalength, SYM.name);
  GetSym();
  if (SYM.sym == SCAN_lbracket)
  { parentry.v.size = 2;
    parentry.v.scalar = false;
    parentry.v.ref = true;
    GetSym();
    accept(SCAN_rbracket, 10);
  }
  Table->enter(parentry, parindex);
  procentry.p.paramsize += parentry.v.size;
  procentry.p.params++;
}

void PARSER::FormalParameters(TABLE_entries &procentry)
// FormalParameters = "(" OneFormal { "," OneFormal } ")" .
{ TABLE_index p;
  GetSym();
  OneFormal(procentry, procentry.p.firstparam);
  while (SYM.sym == SCAN_comma || SYM.sym == SCAN_identifier)
    { accept(SCAN_comma, 13); OneFormal(procentry, p); }
  accept(SCAN_rparen, 17);
}

void PARSER::ProcDeclaration(symset followers)
// ProcDeclaration = ( "PROCEDURE" ProcIdentifier | "FUNCTION" FuncIdentifier )
//                   [ FormalParameters ] ";"
//                   Block ";" .
{ TABLE_entries procentry;
  TABLE_index procindex;
  if (SYM.sym == SCAN_funcsym)
    procentry.idclass = TABLE_funcs;
  else
    procentry.idclass = TABLE_procs;
  GetSym();
  if (SYM.sym != SCAN_identifier)
  { Report->error(6); *procentry.name = '\0'; }
  else
  { sprintf(procentry.name, "%.*s", TABLE_alfalength, SYM.name);
    GetSym();
  }
```

```
  procentry.p.params = 0;
  procentry.p.paramsize = 0;
  procentry.p.firstparam = NULL;
  CGen->storelabel(procentry.p.entrypoint);
  Table->enter(procentry, procindex);
  Table->openscope();
  if (SYM.sym == SCAN_lparen)
  { FormalParameters(procentry);
    Table->update(procentry, procindex);
  }
  test(symset(SCAN_semicolon), followers, 5);
  accept(SCAN_semicolon, 2);
  Block(symset(SCAN_semicolon) + followers, procentry.level + 1,
        procentry.idclass, procentry.p.paramsize + CGEN_headersize);
  accept(SCAN_semicolon, 2);
}

// ++++++++++++++++++++++++ Expressions and Designators++++++++++++++++++

void PARSER::Designator(TABLE_entries entry, symset followers, int errorcode)
// Designator = VarIdentifier [ "[" Expression "]" ] .
{ bool isVariable = (entry.idclass == TABLE_vars);
  if (isVariable)
    CGen->stackaddress(entry.level, entry.v.offset, entry.v.ref);
  else
    Report->error(errorcode);
  GetSym();
  if (SYM.sym == SCAN_lbracket)
  { // subscript
    if (isVariable && entry.v.scalar) Report->error(204);
    GetSym();
    Expression(symset(SCAN_rbracket) + followers);
    if (isVariable)
    { if (entry.v.ref)
        CGen->stackaddress(entry.level, entry.v.offset + 1, entry.v.ref);
      else
        CGen->stackconstant(entry.v.size);
      CGen->subscript();
    }
    accept(SCAN_rbracket, 10);
  }
  else if (isVariable && !entry.v.scalar) Report->error(205);
}

void PARSER::Variable(symset followers, int errorcode)
// Variable = VarDesignator .
// VarDesignator = Designator .
{ TABLE_entries entry;
  bool found;
  if (SYM.sym != SCAN_identifier) { Report->error(6); return; }
  Table->search(SYM.name, entry, found);
  if (!found) Report->error(202);
  Designator(entry, followers, errorcode);
}

void PARSER::ReferenceParameter(void)
{ TABLE_entries entry;
  bool found;
  if (SYM.sym != SCAN_identifier)
    Report->error(214);
  else
  { Table->search(SYM.name, entry, found);
    if (!found)
      Report->error(202);
    else if (entry.idclass != TABLE_vars || entry.v.scalar)
      Report->error(214);
    else
    { CGen->stackaddress(entry.level, entry.v.offset, entry.v.ref);
      // now pass size of array as next parameter
      if (entry.v.ref)
        CGen->stackaddress(entry.level, entry.v.offset + 1, entry.v.ref);
      else
        CGen->stackconstant(entry.v.size);
    }
    GetSym();
  }
}

void PARSER::OneActual(symset followers, TABLE_entries procentry, int &actual)
{ actual++;
  if (Table->isrefparam(procentry, actual))
    ReferenceParameter();
  else
```

```
    Expression(symset(SCAN_comma, SCAN_rparen) + followers);
  test(symset(SCAN_comma, SCAN_rparen), followers - symset(SCAN_identifier), 13);
}

void PARSER::ActualParameters(TABLE_entries procentry, symset followers)
// ActualParameters = [ "(" Expression { "," Expression } ")" ] .
{ int actual = 0;
  if (SYM.sym == SCAN_lparen)
  { GetSym();
    OneActual(followers, procentry, actual);
    while ((SYM.sym == SCAN_comma) || FirstExpression.memb(SYM.sym))
      { accept(SCAN_comma, 13); OneActual(followers, procentry, actual); }
    accept(SCAN_rparen, 17);
  }
  if (actual != procentry.p.params) Report->error(209);
}

void PARSER::Factor(symset followers)
// Factor =   ValDesignator | ConstIdentifier | Number
//          | FuncIdentifier ActualParameters | "(" Expression ")" .
// ValDesignator = Designator .
{ TABLE_entries entry;
  bool found;
  test(FirstFactor, followers, 14);
  switch (SYM.sym)
  { case SCAN_identifier:
      Table->search(SYM.name, entry, found);
      if (!found) Report->error(202);
      switch (entry.idclass)
      { case TABLE_consts:
          CGen->stackconstant(entry.c.value); GetSym(); break;
        case TABLE_funcs:
          GetSym();
          CGen->markstack();
          ActualParameters(entry, followers);
          CGen->call(entry.level, entry.p.entrypoint);
          break;
        default:
          Designator(entry, followers, 206); CGen->dereference(); break;
      }
      break;

    case SCAN_number:
      CGen->stackconstant(SYM.num);
      GetSym();
      break;

    case SCAN_lparen:
      GetSym();
      Expression(symset(SCAN_rparen) + followers);
      accept(SCAN_rparen, 17);
      break;

    case SCAN_stringsym:
      Report->error(14);
      GetSym();
      break;

    default:
      Report->error(14);
      break;
  }
}

void PARSER::Term(symset followers)
// Term = Factor { ( "*" | "/" ) Factor } .
{ SCAN_symtypes opsym;
  Factor(symset(SCAN_times, SCAN_slash) + followers);
  while (SYM.sym == SCAN_times || SYM.sym == SCAN_slash || FirstFactor.memb(SYM.sym))
  { if (SYM.sym == SCAN_times || SYM.sym == SCAN_slash)
      { opsym = SYM.sym; GetSym(); }
    else
      { opsym = SCAN_times; Report->error(20); }
    Factor(symset(SCAN_times, SCAN_slash) + followers);
    CGen->binaryintegerop(op(opsym));
  }
}

void PARSER::Expression(symset followers)
// Expression = [ "+" | "-" ] Term { ( "+" | "-" ) Term } .
{ SCAN_symtypes opsym;
  if (SYM.sym == SCAN_plus)
  { GetSym();
```

```
    Term(symset(SCAN_plus, SCAN_minus) + followers);
  }
  else if (SYM.sym == SCAN_minus)
  { GetSym();
    Term(symset(SCAN_plus, SCAN_minus) + followers);
    CGen->negateinteger();
  }
  else
    Term(symset(SCAN_plus, SCAN_minus) + followers);
  while (SYM.sym == SCAN_plus || SYM.sym == SCAN_minus)
  { opsym = SYM.sym; GetSym();
    Term(symset(SCAN_plus, SCAN_minus) + followers);
    CGen->binaryintegerop(op(opsym));
  }
}

void PARSER::Condition(symset followers)
// Condition = Expression Relop Expression .
{ SCAN_symtypes opsym;
  symset stopset = RelOpSyms + followers;
  Expression(stopset);
  if (!RelOpSyms.memb(SYM.sym)) { Report->error(19); return; }
  opsym = SYM.sym; GetSym();
  Expression(followers); CGen->comparison(op(opsym));
}

// ++++++++++++++++++++++++ Statement Part ++++++++++++++++++++++++++

void PARSER::CompoundStatement(symset followers)
// CompoundStatement = "BEGIN" Statement { ";" Statement } "END" .
{ accept(SCAN_beginsym, 34);
  Statement(symset(SCAN_semicolon, SCAN_endsym) + followers);
  while (SYM.sym == SCAN_semicolon || FirstStatement.memb(SYM.sym))
    { accept(SCAN_semicolon, 2);
      Statement(symset(SCAN_semicolon, SCAN_endsym) + followers);
    }
  accept(SCAN_endsym, 24);
}

void PARSER::Assignment(symset followers, TABLE_entries entry)
// Assignment = VarDesignator ":=" Expression .
// VarDesignator = Designator .
{ Designator(entry, symset(SCAN_becomes, SCAN_eqlsym) + followers, 210);
  if (SYM.sym == SCAN_becomes)
    GetSym();
  else
    { Report->error(21); if (SYM.sym == SCAN_eqlsym) GetSym(); }
  Expression(followers);
  CGen->assign();
}

void PARSER::ProcedureCall(symset followers, TABLE_entries entry)
// ProcedureCall = ProcIdentifier ActualParameters .
{ GetSym();
  CGen->markstack();
  ActualParameters(entry, followers);
  CGen->call(entry.level, entry.p.entrypoint);
}

void PARSER::IfStatement(symset followers)
// IfStatement = "IF" Condition "THEN" Statement .
{ CGEN_labels testlabel;

  GetSym();
  Condition(symset(SCAN_thensym, SCAN_dosym) + followers);
  CGen->jumponfalse(testlabel, CGen->undefined);
  if (SYM.sym == SCAN_thensym)
    GetSym();
  else
    { Report->error(23); if (SYM.sym == SCAN_dosym) GetSym(); }
  Statement(followers);
  CGen->backpatch(testlabel);
}

void PARSER::WhileStatement(symset followers)
// WhileStatement = "WHILE" Condition "DO" Statement .
{ CGEN_labels startloop, testlabel, dummylabel;
  GetSym();
  CGen->storelabel(startloop);
  Condition(symset(SCAN_dosym) + followers);
  CGen->jumponfalse(testlabel, CGen->undefined);
  accept(SCAN_dosym, 25);
  Statement(followers);
```

```
  CGen->jump(dummylabel, startloop);
  CGen->backpatch(testlabel);
}

void PARSER::ReturnStatement(symset followers)
// ReturnStatement = "RETURN" [ Expression ]
{ GetSym();
  switch (blockclass)
  { case TABLE_funcs:
      if (!FirstExpression.memb(SYM.sym)) Report->error(220); // an Expression is mandato
      else
      { CGen->stackaddress(blocklevel, 1, false);
        Expression(followers);
        CGen->assign();
        CGen->leavefunction(blocklevel);
      }
      break;

    case TABLE_procs:   // simple return
      CGen->leaveprocedure(blocklevel);
      if (FirstExpression.memb(SYM.sym)) // we may NOT have an Expression
        { Report->error(219); Expression(followers); }
      break;

    case TABLE_progs:   // okay in main program - just halts
      CGen->leaveprogram();
      if (FirstExpression.memb(SYM.sym)) // we may NOT have an Expression
        { Report->error(219); Expression(followers); }
      break;
  }
}

void PARSER::WriteElement(symset followers)
// WriteElement = Expression | String .
{ CGEN_labels startstring;
  if (SYM.sym != SCAN_stringsym)
  { Expression(symset(SCAN_comma, SCAN_rparen) + followers);
    CGen->writevalue();
  }
  else
  { CGen->stackstring(SYM.name, startstring);
    CGen->writestring(startstring);
    GetSym();
  }
}

void PARSER::WriteStatement(symset followers)
// WriteStatement = "WRITE" [ "(" WriteElement { "," WriteElement } ")" ] .
{ GetSym();
  if (SYM.sym == SCAN_lparen)
  { GetSym();
    WriteElement(followers);
    while (SYM.sym == SCAN_comma || FirstExpression.memb(SYM.sym))
    { accept(SCAN_comma, 13); WriteElement(followers); }
    accept(SCAN_rparen, 17);
  }
  CGen->newline();
}

void PARSER::ReadStatement(symset followers)
// ReadStatement = "READ" "(" Variable { "," Variable } ")" .
{ GetSym();
  if (SYM.sym != SCAN_lparen) { Report->error(18); return; }
  GetSym();
  Variable(symset(SCAN_comma, SCAN_rparen) + followers, 211);
  CGen->readvalue();
  while (SYM.sym == SCAN_comma || SYM.sym == SCAN_identifier)
  { accept(SCAN_comma, 13);
    Variable(symset(SCAN_comma, SCAN_rparen) + followers, 211);
    CGen->readvalue();
  }
  accept(SCAN_rparen, 17);
}

void PARSER::ProcessCall(symset followers, int &processes)
// ProcessCall = ProcIdentifier ActualParameters .
{ TABLE_entries entry;
  bool found;
  if (!FirstStatement.memb(SYM.sym)) return;
  if (SYM.sym != SCAN_identifier)
    { Report->error(217); Statement(followers); return; } // recovery
  Table->search(SYM.name, entry, found);
  if (!found) Report->error(202);
```

```
  if (entry.idclass != TABLE_procs)
    { Report->error(217); Statement(followers); return; } // recovery
  GetSym();
  CGen->markstack();
  ActualParameters(entry, followers);
  CGen->forkprocess(entry.p.entrypoint);
  processes++;
}

void PARSER::CobeginStatement(symset followers)
// CobeginStatement := "COBEGIN" ProcessCall { ";" ProcessCall } "COEND"
// count number of processes
{ int processes = 0;
  CGEN_labels start;
  if (blockclass != TABLE_progs) Report->error(215);  // only from global level
  GetSym(); CGen->cobegin(start);
  ProcessCall(symset(SCAN_semicolon, SCAN_coendsym) + followers, processes);
  while (SYM.sym == SCAN_semicolon || FirstStatement.memb(SYM.sym))
    { accept(SCAN_semicolon, 2);
      ProcessCall(symset(SCAN_semicolon, SCAN_coendsym) +  followers, processes);
    }
  CGen->coend(start, processes);
  accept(SCAN_coendsym, 38);
}

void PARSER::SemaphoreStatement(symset followers)
// SemaphoreStatement = ("WAIT" | "SIGNAL")  "(" VarDesignator ")" .
{ bool wait = (SYM.sym == SCAN_waitsym);
  GetSym();
  if (SYM.sym != SCAN_lparen) { Report->error(18); return; }
  GetSym();
  Variable(symset(SCAN_rparen) + followers, 206);
  if (wait) CGen->waitop(); else CGen->signalop();
  accept(SCAN_rparen, 17);
}

void PARSER::Statement(symset followers)
// Statement = [ CompoundStatement | Assignment | ProcedureCall
//               | IfStatement | WhileStatement | ReturnStatement
//               | WriteStatement | ReadStatement | CobeginStatement
//               | WaitStatement | SignalStatement | "STACKDUMP" ] .
{ TABLE_entries entry;
  bool found;
  if (FirstStatement.memb(SYM.sym))
  { switch (SYM.sym)
    { case SCAN_identifier:
        Table->search(SYM.name, entry, found);
        if (!found) Report->error(202);
        if (entry.idclass == TABLE_procs)
          ProcedureCall(followers, entry);
        else
          Assignment(followers, entry);
        break;
      case SCAN_ifsym:     IfStatement(followers); break;
      case SCAN_whilesym:  WhileStatement(followers); break;
      case SCAN_returnsym: ReturnStatement(followers); break;
      case SCAN_writesym:  WriteStatement(followers); break;
      case SCAN_readsym:   ReadStatement(followers); break;
      case SCAN_beginsym:  CompoundStatement(followers); break;
      case SCAN_stacksym:  CGen->dump(); GetSym(); break;
      case SCAN_cobegsym:  CobeginStatement(followers); break;
      case SCAN_waitsym:
      case SCAN_signalsym: SemaphoreStatement(followers); break;
    }
  }
  // test(Followers - symset(SCAN_identifier), EmptySet, 32) or
  test(followers, EmptySet, 32);
}

void PARSER::Block(symset followers, int blklevel, TABLE_idclasses blkclass,
                   int initialframesize)
// Block = { ConstDeclarations | VarDeclarations | ProcDeclaration }
//         CompoundStatement .
{ int framesize = initialframesize;  // activation record space
  CGEN_labels entrypoint;
  CGen->jump(entrypoint, CGen->undefined);
  test(FirstBlock, followers, 3);
  if (blklevel > CGEN_levmax) Report->error(213);
  do
  { if (SYM.sym == SCAN_constsym) ConstDeclarations();
    if (SYM.sym == SCAN_varsym) VarDeclarations(framesize);
    while (SYM.sym == SCAN_procsym || SYM.sym == SCAN_funcsym)
      ProcDeclaration(followers);
```

```
    test(FirstBlock, followers, 4);
  } while (FirstDeclaration.memb(SYM.sym));
  // blockclass, blocklevel global for efficiency
  blockclass = blkclass;
  blocklevel = blklevel;
  CGen->backpatch(entrypoint);   // reserve space for variables
  CGen->openstackframe(framesize - initialframesize);
  CompoundStatement(followers);
  switch (blockclass)
  { case TABLE_progs: CGen->leaveprogram(); break;
    case TABLE_procs: CGen->leaveprocedure(blocklevel); break;
    case TABLE_funcs: CGen->functioncheck(); break;
  }
  test(followers, EmptySet, 35);
  if (debugging) Table->printtable(Srce->lst);  // demonstration purposes
  Table->closescope();
}

void PARSER::ClangProgram(void)
// ClangProgram = "PROGRAM" ProgIdentifier ";" Block "." .
{ TABLE_entries progentry;
  TABLE_index progindex;
  accept(SCAN_progsym, 36);
  if (SYM.sym != SCAN_identifier)
    Report->error(6);
  else
  { sprintf(progentry.name, "%.*s", TABLE_alfalength, SYM.name);
    debugging = (strcmp(SYM.name, "DEBUG") == 0);
    progentry.idclass = TABLE_progs;
    Table->enter(progentry, progindex);
    GetSym();
  }
  Table->openscope();
  accept(SCAN_semicolon, 2);
  Block(symset(SCAN_period, SCAN_eofsym) + FirstBlock + FirstStatement,
                progentry.level + 1, TABLE_progs, 0);
  accept(SCAN_period, 37);
}

void PARSER::parse(void)
{ GetSym(); ClangProgram(); }


----- table.h ----------------------------------------------------------------

// Handle symbol table for Clang level 3/4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency
// P.D. Terry, Rhodes University, 1996

#ifndef TABLE_H
#define TABLE_H

#include "cgen.h"
#include "report.h"

const int TABLE_alfalength = 15; // maximum length of identifiers
typedef char TABLE_alfa[TABLE_alfalength + 1];

enum TABLE_idclasses
{ TABLE_consts, TABLE_vars, TABLE_progs, TABLE_procs, TABLE_funcs };

struct TABLE_nodes;
typedef TABLE_nodes *TABLE_index;

struct TABLE_entries {
  TABLE_alfa name;             // identifier
  int level;                   // static level
  TABLE_idclasses idclass;     // class
  union {
    struct {
      int value;
    } c;                       // constants
    struct {
      int size, offset;
      bool ref, scalar;
    } v;                       // variables
    struct {
      int params, paramsize;
      TABLE_index firstparam;
      CGEN_labels entrypoint;
    } p;                       // procedures, functions
  };
};
```

```
struct TABLE_nodes {
  TABLE_entries entry;
  TABLE_index next;
};

struct SCOPE_nodes {
  SCOPE_nodes *down;
  TABLE_index first;
};

class TABLE {
  public:
    void openscope(void);
    // Opens new scope for a new block

    void closescope(void);
    // Closes scope at block exit

    void enter(TABLE_entries &entry, TABLE_index &position);
    // Adds entry to symbol table, and returns its position

    void search(char *name, TABLE_entries &entry, bool &found);
    // Searches table for presence of name.  If found then returns entry

    void update(TABLE_entries &entry, TABLE_index position);
    // Updates entry at known position

    bool isrefparam(TABLE_entries &procentry, int n);
    // Returns true if nth parameter for procentry is passed by reference

    void printtable(FILE *lst);
    // Prints symbol table for diagnostic purposes

    TABLE(REPORT *R);
    // Initializes symbol table

  private:
    TABLE_index sentinel;
    SCOPE_nodes *topscope;
    REPORT *Report;
    int currentlevel;
};

#endif /*TABLE_H*/


----- table.cpp ----------------------------------------------------------------

// Handle symbol table for Clang level 3/4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "table.h"

void TABLE::openscope(void)
{ SCOPE_nodes *newscope = new SCOPE_nodes;
  newscope->down = topscope; newscope->first = sentinel;
  topscope = newscope;
  currentlevel++;
}

void TABLE::closescope(void)
{ SCOPE_nodes *old = topscope;
  topscope = topscope->down; delete old;
  currentlevel--;
}

void TABLE::enter(TABLE_entries &entry, TABLE_index &position)
{ TABLE_index look = topscope->first;
  TABLE_index last = NULL;
  position = new TABLE_nodes;
  sprintf(sentinel->entry.name, "%.*s", TABLE_alfalength, entry.name);
  while (strcmp(look->entry.name, sentinel->entry.name))
  { last = look; look = look->next; }
  if (look != sentinel) Report->error(201);
  entry.level = currentlevel;
  position->entry = entry; position->next = look;
  if (!last) topscope->first = position; else last->next = position;
}

void TABLE::update(TABLE_entries &entry, TABLE_index position)
```

```
{ position->entry = entry; }

void TABLE::search(char *name, TABLE_entries &entry, bool &found)
{ TABLE_index look;
  SCOPE_nodes *scope = topscope;
  sprintf(sentinel->entry.name, "%.*s", TABLE_alfalength, name);
  while (scope)
  { look = scope->first;
    while (strcmp(look->entry.name, sentinel->entry.name)) look = look->next;
    if (look != sentinel) { found = true; entry = look->entry; return; }
    scope = scope->down;
  }
  found = false; entry = sentinel->entry;
}

bool TABLE::isrefparam(TABLE_entries &procentry, int n)
{ if (n > procentry.p.params) return false;
  TABLE_index look = procentry.p.firstparam;
  while (n > 1) { look = look->next; n--; }
  return look->entry.v.ref;
}

void TABLE::printtable(FILE *lst)
{ SCOPE_nodes *scope = topscope;
  TABLE_index current;
  putc('\n', lst);
  while (scope)
  { current = scope->first;
    while (current != sentinel)
    { fprintf(lst, "%-16s", current->entry.name);
      switch (current->entry.idclass)
      { case TABLE_consts:
          fprintf(lst, " Constant  %7d\n", current->entry.c.value);
          break;

        case TABLE_vars:
          fprintf(lst, " Variable  %3d%4d%4d\n",
                  current->entry.level, current->entry.v.offset,
                  current->entry.v.size);
          break;

        case TABLE_procs:
          fprintf(lst, " Procedure %3d%4d%4d\n",
                  current->entry.level, current->entry.p.entrypoint,
                  current->entry.p.params);
          break;

        case TABLE_funcs:
          fprintf(lst, " Function  %3d%4d%4d\n",
                  current->entry.level, current->entry.p.entrypoint,
                  current->entry.p.params);
          break;

       case TABLE_progs:
          fprintf(lst, " Program   \n");
          break;
      }
      current = current->next;
    }
    scope = scope->down;
  }
}

TABLE::TABLE(REPORT *R)
{ sentinel = new TABLE_nodes;
  sentinel->entry.name[0] = '\0'; sentinel->entry.level = 0;
  sentinel->entry.idclass = TABLE_progs;
  currentlevel = 0; topscope = NULL;   // for level 0 identifiers
  Report = R; openscope(); currentlevel = 0;
}


----- cgen.h ----------------------------------------------------------------

// Code Generation for Clang level 4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#ifndef CGEN_H
#define CGEN_H

#include "misc.h"
```

```
#include "stkmc.h"
#include "report.h"

#define CGEN_headersize  STKMC_headersize
#define CGEN_levmax      STKMC_levmax

enum CGEN_operators {
  CGEN_opadd, CGEN_opsub, CGEN_opmul, CGEN_opdvd, CGEN_opeql, CGEN_opneq,
  CGEN_oplss, CGEN_opgeq, CGEN_opgtr, CGEN_opleq
};

typedef short CGEN_labels;
typedef char  CGEN_levels;

class CGEN {
  public:
    CGEN_labels undefined;    // for forward references

    CGEN(REPORT *R);
    // Initializes code generator

    void negateinteger(void);
    // Generates code to negate integer value on top of evaluation stack

    void binaryintegerop(CGEN_operators op);
    // Generates code to pop two values A,B from evaluation stack
    // and push value A op B

    void comparison(CGEN_operators op);
    // Generates code to pop two integer values A,B from stack
    // and push Boolean value A OP B

    void readvalue(void);
    // Generates code to read value; store on address found on top of stack

    void writevalue(void);
    // Generates code to output value from top of stack

    void newline(void);
    // Generates code to output line mark

    void writestring(CGEN_labels location);
    // Generates code to output string stored at known location

    void stackstring(char *str, CGEN_labels &location);
    // Stores str in literal pool in memory and return its location

    void stackconstant(int number);
    // Generates code to push number onto evaluation stack

    void stackaddress(int level, int offset, bool indirect);
    // Generates code to push address for known level, offset onto evaluation stack.
    // Addresses of reference parameters are treated as indirect

    void subscript(void);
    // Generates code to index an array and check that bounds are not exceeded

    void dereference(void);
    // Generates code to replace top of evaluation stack by the value found at the
    // address currently stored on top of the stack

    void assign(void);
    // Generates code to store value currently on top-of-stack on the address
    // given by next-to-top, popping these two elements

    void openstackframe(int size);
    // Generates code to reserve space for size variables

    void leaveprogram(void);
    // Generates code needed to leave a program (halt)

    void leaveprocedure(int blocklevel);
    // Generates code needed to leave a regular procedure at given blocklevel

    void leavefunction(int blocklevel);
    // Generates code needed as we leave a function at given blocklevel

    void functioncheck(void);
    // Generate code to ensure that a function has returned a value

    void cobegin(CGEN_labels &location);
    // Generates code to initiate concurrent processing
```

```
    void coend(CGEN_labels location, int number);
    // Generates code to terminate concurrent processing

    void storelabel(CGEN_labels &location);
    // Stores address of next instruction in location for use in backpatching

    void jump(CGEN_labels &here, CGEN_labels destination);
    // Generates unconditional branch from here to destination

    void jumponfalse(CGEN_labels &here, CGEN_labels destination);
    // Generates branch from here to destination, conditional on the Boolean
    // value currently on top of the evaluation stack, popping this value

    void backpatch(CGEN_labels location);
    // Stores the current location counter as the address field of the branch
    // instruction currently held in an incomplete form at location

    void markstack(void);
    // Generates code to reserve mark stack storage before calling procedure

    void call(int level, CGEN_labels entrypoint);
    // Generates code to enter procedure at known level and entrypoint

    void forkprocess(CGEN_labels entrypoint);
    // Generates code to initiate process at known entrypoint

    void signalop(void);
    // Generates code for semaphore signalling operation

    void waitop(void);
    // Generates code for semaphore wait operation

    void dump(void);
    // Generates code to dump the current state of the evaluation stack

    void getsize(int &codelength, int &initsp);
    // Returns length of generated code and initial stack pointer

    int gettop(void);
    // Return codetop

    void emit(int word);
    // Emits single word

  private:
    REPORT *Report;
    bool generatingcode;
    STKMC_address codetop, stktop;
};

#endif /*CGEN_H*/


----- cgen.cpp ----------------------------------------------------------------

// Code Generation for Clang Level 4 compiler/interpreter
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#include "cgen.h"

extern STKMC* Machine;

CGEN::CGEN(REPORT *R)
{ undefined = 0;    // for forward references (exported)
  Report = R;
  generatingcode = true;
  codetop = 0;
  stktop = STKMC_memsize - 1;
}

void CGEN::emit(int word)
// Code generator for single word
{ if (!generatingcode) return;
  if (codetop >= stktop)
    { Report->error(212); generatingcode = false; }
  else
    { Machine->mem[codetop] = word; codetop++; }
}

void CGEN::negateinteger(void)
{ emit(int(STKMC_neg)); }
```

```
void CGEN::binaryintegerop(CGEN_operators op)
{ switch (op)
  { case CGEN_opmul:  emit(int(STKMC_mul)); break;
    case CGEN_opdvd:  emit(int(STKMC_dvd)); break;
    case CGEN_opadd:  emit(int(STKMC_add)); break;
    case CGEN_opsub:  emit(int(STKMC_sub)); break;
  }
}

void CGEN::comparison(CGEN_operators op)
{ switch (op)
  { case CGEN_opeql:  emit(int(STKMC_eql)); break;
    case CGEN_opneq:  emit(int(STKMC_neq)); break;
    case CGEN_oplss:  emit(int(STKMC_lss)); break;
    case CGEN_opleq:  emit(int(STKMC_leq)); break;
    case CGEN_opgtr:  emit(int(STKMC_gtr)); break;
    case CGEN_opgeq:  emit(int(STKMC_geq)); break;
  }
}

void CGEN::readvalue(void)
{ emit(int(STKMC_inn)); }

void CGEN::writevalue(void)
{ emit(int(STKMC_prn)); }

void CGEN::newline(void)
{ emit(int(STKMC_nln)); }

void CGEN::writestring(CGEN_labels location)
{ emit(int(STKMC_prs)); emit(location); }

void CGEN::stackstring(char *str, CGEN_labels &location)
{ int l = strlen(str);
  if (stktop <= codetop + l + 1)
    { Report->error(212); generatingcode = false; return; }
  location = stktop - 1;
  for (int i = 0; i < l; i++) { stktop--; Machine->mem[stktop] = str[i]; }
  stktop--; Machine->mem[stktop] = 0;
}

void CGEN::stackconstant(int number)
{ emit(int(STKMC_lit)); emit(number); }

void CGEN::stackaddress(int level, int offset, bool indirect)
{ emit(int(STKMC_adr)); emit(level); emit(-offset);
  if (indirect) emit(int(STKMC_val));
}

void CGEN::subscript(void)
{ emit(int(STKMC_ind)); }

void CGEN::dereference(void)
{ emit(int(STKMC_val)); }

void CGEN::assign(void)
{ emit(int(STKMC_sto)); }

void CGEN::openstackframe(int size)
{ if (size > 0) { emit(int(STKMC_dsp)); emit(size); } }

void CGEN::leaveprogram(void)
{ emit(int(STKMC_hlt)); }

void CGEN::leavefunction(int blocklevel)
{ emit(int(STKMC_ret)); emit(blocklevel); emit(1); }

void CGEN::functioncheck(void)
{ emit(int(STKMC_nfn)); }

void CGEN::leaveprocedure(int blocklevel)
{ emit(int(STKMC_ret)); emit(blocklevel); emit(0); }

void CGEN::cobegin(CGEN_labels &location)
{ location = codetop; emit(int(STKMC_cbg)); emit(undefined); }

void CGEN::coend(CGEN_labels location, int number)
{ if (number >= STKMC_procmax) Report->error(216);
  else { Machine->mem[location+1] = number; emit(int(STKMC_cnd)); }
}

void CGEN::storelabel(CGEN_labels &location)
```

```
{ location = codetop; }

void CGEN::jump(CGEN_labels &here, CGEN_labels destination)
{ here = codetop; emit(int(STKMC_brn)); emit(destination); }

void CGEN::jumponfalse(CGEN_labels &here, CGEN_labels destination)
{ here = codetop; emit(int(STKMC_bze)); emit(destination); }

void CGEN::backpatch(CGEN_labels location)
{ if (codetop == location + 2 &&
      STKMC_opcodes(Machine->mem[location]) == STKMC_brn)
    codetop -= 2;
  else
    Machine->mem[location+1] = codetop;
}

void CGEN::markstack(void)
{ emit(int(STKMC_mst)); }

void CGEN::forkprocess(CGEN_labels entrypoint)
{ emit(int(STKMC_frk)); emit(entrypoint); }

void CGEN::call(int level, CGEN_labels entrypoint)
{ emit(int(STKMC_cal)); emit(level); emit(entrypoint); }

void CGEN::signalop(void)
{ emit(int(STKMC_sig)); }

void CGEN::waitop(void)
{ emit(int(STKMC_wgt)); }

void CGEN::dump(void)
{ emit(int(STKMC_stk)); }

void CGEN::getsize(int &codelength, int &initsp)
{ codelength = codetop; initsp = stktop; }

int CGEN::gettop(void)
{ return codetop; }


----- stkmc.h ----------------------------------------------------------------

// Definition of simple stack machine and simple emulator for Clang level 4
// Includes procedures, functions, parameters, arrays, concurrency.
// This version emulates one CPU time sliced between processes.
// Display machine
// P.D. Terry, Rhodes University, 1996

#ifndef STKMC_H
#define STKMC_H

#define STKMC_version     "Clang 4.0"
const int STKMC_memsize = 512;  // Limit on memory
const int STKMC_levmax = 5;     // Limit on Display
const int STKMC_headersize = 5; // Size of minimum activation record
const int STKMC_procmax = 10;   // Limit on concurrent processes

// machine instructions - order important
enum STKMC_opcodes {
  STKMC_cal, STKMC_ret, STKMC_adr, STKMC_frk, STKMC_cbg, STKMC_lit, STKMC_dsp,
  STKMC_brn, STKMC_bze, STKMC_prs, STKMC_wgt, STKMC_sig, STKMC_cnd, STKMC_nfn,
  STKMC_mst, STKMC_add, STKMC_sub, STKMC_mul, STKMC_dvd, STKMC_eql, STKMC_neq,
  STKMC_lss, STKMC_geq, STKMC_gtr, STKMC_leq, STKMC_neg, STKMC_val, STKMC_sto,
  STKMC_ind, STKMC_stk, STKMC_hlt, STKMC_inn, STKMC_prn, STKMC_nln, STKMC_nop,
  STKMC_nul
};

typedef enum {
  running, finished, badmem, baddata, nodata, divzero, badop, badind,
  badfun, badsem, deadlock
} status;
typedef int STKMC_address;
typedef int STKMC_levels;
typedef int STKMC_procindex;

class STKMC {
  public:
    int mem[STKMC_memsize];  // virtual machine memory

    void listcode(char *filename, STKMC_address codelen);
    // Lists the codelen instructions stored in mem on named output file
```

```
    void emulator(STKMC_address initpc, STKMC_address codelen,
                  STKMC_address initsp, FILE *data, FILE *results,
                  bool tracing);
    // Emulates action of the codelen instructions stored in mem, with
    // program counter initialized to initpc, stack pointer initialized to
    // initsp.  data and results are used for I/O.  Tracing at the code level
    // may be requested

    void interpret(STKMC_address codelen, STKMC_address initsp);
    // Interactively opens data and results files.  Then interprets the
    // codelen instructions stored in mem, with stack pointer initialized
    // to initsp

    STKMC_opcodes opcode(char *str);
    // Maps str to opcode, or to MC_nul if no match can be found

    STKMC();
    // Initializes stack machine

  private:
    struct processor {
      STKMC_opcodes ir;  // Instruction register
      int bp;            // Base pointer
      int sp;            // Stack pointer
      int mp;            // Mark Stack pointer
      int pc;            // Program counter
    };
    struct processrec { // Process descriptors
      STKMC_address bp, mp, sp, pc;      // Shadow registers
      STKMC_procindex next;              // Ring pointer
      STKMC_procindex queue;             // Linked, waiting on semaphore
      bool ready;                        // Process ready flag
      STKMC_address stackmax, stackmin;  // Memory limits
      int display[STKMC_levmax];         // Display registers
    };
    processor cpu;
    status ps;

    bool inbounds(int p);

    char *mnemonics[STKMC_nul+1];
    void stackdump(STKMC_address initsp, FILE *results, STKMC_address pcnow);
    void trace(FILE *results, STKMC_address pcnow);
    void postmortem(FILE *results, STKMC_address pcnow);

    int slice;
    STKMC_procindex current, nexttorun;
    processrec process[STKMC_procmax + 1];
    void swapregisters(void);
    void chooseprocess(void);
    void signal(STKMC_address semaddress);
    void wait(STKMC_address semaddress);
};

#endif /*STKMC_H*/


----- stkmc.cpp ------------------------------------------------------------

// Definition of simple stack machine and simple emulator for Clang level 4
// Includes procedures, functions, parameters, arrays, concurrency.
// This version emulates one CPU time sliced between processes.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "stkmc.h"
#include <time.h>

#define random(num) (rand() % (num))
#define BLANKS "                                                  "

const int maxslice      = 8;  // maximum time slice
const int processreturn = 0;  // fictitious return address

STKMC_opcodes STKMC::opcode(char *str)
{ STKMC_opcodes l = STKMC_opcodes(0);
  for (int i = 0; str[i]; i++) str[i] = toupper(str[i]);
  while (l != STKMC_nul && strcmp(str, mnemonics[l]))
    l = STKMC_opcodes(long(l) + 1);
  return l;
}
```

```
void STKMC::listcode(char *filename, STKMC_address codelen)
{ STKMC_address i, j;
  STKMC_opcodes op;
  if (*filename == '\0') return;
  FILE *codefile = fopen(filename, "w");
  if (codefile == NULL) return;
/* The following may be useful for debugging the interpreter
  i = 0;
  while (i < codelen)
  { fprintf(codefile, "%4d", mem[i]);
    if ((i + 1) % 16 == 0) putc('\n', codefile);
    i++;
  }
  putc('\n', codefile);
*/
  i = 0;
  while (i < codelen)
  { op = STKMC_opcodes(mem[i] % (int(STKMC_nul) + 1)); // force in range
    fprintf(codefile, "%10d %s", i, mnemonics[op]);
    switch (op)
    { case STKMC_cal:
      case STKMC_ret:
      case STKMC_adr:
        i = (i + 1) % STKMC_memsize; fprintf(codefile, "%3d", mem[i]);
        i = (i + 1) % STKMC_memsize; fprintf(codefile, "%6d", mem[i]);
        break;

      case STKMC_frk:
      case STKMC_cbg:
      case STKMC_lit:
      case STKMC_dsp:
      case STKMC_brn:
      case STKMC_bze:
        i = (i + 1) % STKMC_memsize; fprintf(codefile, "%9d", mem[i]);
        break;

      case STKMC_prs:
        i = (i + 1) % STKMC_memsize;
        j = mem[i]; fprintf(codefile, "   '");
        while (mem[j] != 0) { putc(mem[j], codefile); j--; }
        putc('\'', codefile);
        break;
    }
    i = (i + 1) % STKMC_memsize;
    putc('\n', codefile);
  }
  fclose(codefile);
}

void STKMC::swapregisters(void)
// Save current machine registers; restore from next process
{ process[current].bp = cpu.bp;   cpu.bp = process[nexttorun].bp;
  process[current].mp = cpu.mp;   cpu.mp = process[nexttorun].mp;
  process[current].sp = cpu.sp;   cpu.sp = process[nexttorun].sp;
  process[current].pc = cpu.pc;   cpu.pc = process[nexttorun].pc;
}

void STKMC::chooseprocess(void)
// From current process, traverse ring of descriptors to next ready process
{ if (slice != 0) { slice--; return; }
  do { nexttorun = process[nexttorun].next; } while (!process[nexttorun].ready);
  if (nexttorun != current) swapregisters();
  slice = random(maxslice) + 3;
}

bool STKMC::inbounds(int p)
// Check that memory pointer P does not go out of bounds.  This should not
// happen with correct code, but it is just as well to check
{ if (p < process[current].stackmin || p >= STKMC_memsize) ps = badmem;
  return (ps == running);
}

void STKMC::stackdump(STKMC_address initsp, FILE *results, STKMC_address pcnow)
// Dump data area - useful for debugging
{ int online = 0;
  fprintf(results, "\nStack dump at %4d CPU:%4d", pcnow, current);
  fprintf(results, " SP:%4d BP:%4d", cpu.sp, cpu.bp);
  fprintf(results, " SM:%4d", process[current].stackmin);
  if (cpu.bp < initsp) fprintf(results, " Return Address:%4d", mem[cpu.bp - 4]);
  putc('\n', results);
  for (int l = process[current].stackmax - 1; l >= cpu.sp; l--)
  { fprintf(results, "%7d:%5d", l, mem[l]);
    online++; if (online % 6 == 0) putc('\n', results);
```

```
  }
  fprintf(results, "\nDisplay");
  for (l = 0; l < STKMC_levmax; l++)
    fprintf(results, "%4d", process[current].display[l]);
  putc('\n', results);
}

void STKMC::trace(FILE *results, STKMC_address pcnow)
// Simple trace facility for run time debugging
{ fprintf(results, "CPU:%4d PC:%4d BP:%4d", current, pcnow, cpu.bp);
  fprintf(results, " SP:%4d TOS:", cpu.sp);
  if (cpu.sp < STKMC_memsize)
    fprintf(results, "%4d", mem[cpu.sp]);
  else
    fprintf(results, "????");
  fprintf(results, " %s", mnemonics[cpu.ir]);
  switch (cpu.ir)
  { case STKMC_cal:
    case STKMC_ret:
    case STKMC_adr:
      fprintf(results, "%3d%6d", mem[cpu.pc], mem[cpu.pc + 1]);
      break;
    case STKMC_frk:
    case STKMC_cbg:
    case STKMC_lit:
    case STKMC_dsp:
    case STKMC_brn:
    case STKMC_bze:
    case STKMC_prs:
      fprintf(results, "%9d", mem[cpu.pc]); break;
    // no default needed
  }
  putc('\n', results);
}

void STKMC::postmortem(FILE *results, int pcnow)
// Report run time error and position
{ putc('\n', results);
  switch (ps)
  { case badop:    fprintf(results, "Illegal opcode"); break;
    case nodata:   fprintf(results, "No more data"); break;
    case baddata:  fprintf(results, "Invalid data"); break;
    case divzero:  fprintf(results, "Division by zero"); break;
    case badmem:   fprintf(results, "Memory violation"); break;
    case badind:   fprintf(results, "Subscript out of range"); break;
    case badfun:   fprintf(results, "Function did not return value"); break;
    case badsem:   fprintf(results, "Bad Semaphore operation"); break;
    case deadlock: fprintf(results, "Deadlock"); break;
  }
  fprintf(results, " at %4d in process %d\n", pcnow, current);
}

void STKMC::signal(STKMC_address semaddress)
{ if (mem[semaddress] >= 0)                 // do we need to waken a process?
  { mem[semaddress]++; return; }            // no - simply increment semaphore
  STKMC_procindex woken = -mem[semaddress]; // negate to find index
  mem[semaddress] = -process[woken].queue;  // bump queue pointer
  process[woken].queue = 0;                 // remove from queue
  process[woken].ready = true;              // and allow to be reactivated
}

void STKMC::wait(STKMC_address semaddress)
{ STKMC_procindex last, now;
  if (mem[semaddress] > 0)                  // do we need to suspend?
  { mem[semaddress]--; return; }            // no - simply decrement semaphore
  slice = 0; chooseprocess();               // choose the next process
  process[current].ready = false;           // and suspend this one
  if (current == nexttorun) { ps = deadlock; return; }
  now = -mem[semaddress];                   // look for end of semaphore queue
  while (now != 0) { last = now; now = process[now].queue; }
  if (mem[semaddress] == 0)
    mem[semaddress] = -current;             // first in queue
  else
    process[last].queue = current;          // place at end of existing queue
  process[current].queue = 0;               // and mark as the new end of queue
}

void STKMC::emulator(STKMC_address initpc, STKMC_address codelen,
                     STKMC_address initsp, FILE *data, FILE *results,
                     bool tracing)
{ STKMC_address pcnow;       // Current program counter
  STKMC_address parentsp;    // Original stack pointer of parent
  STKMC_procindex nprocs;    // Number of concurrent processes
```

```
int partition;             // Memory allocated to each process
int loop;
srand(time(NULL));         // Initialize random number generator
process[0].stackmax = initsp;
process[0].stackmin = codelen;
process[0].queue = 0;
process[0].ready = true;
cpu.sp = initsp;
cpu.bp = initsp;   // initialize registers
cpu.pc = initpc;   // initialize program counter
for (int l = 0; l < STKMC_levmax; l++) process[0].display[l] = initsp;
nexttorun = 0;
nprocs = 0;
slice = 0;
ps = running;
do
{ pcnow = cpu.pc; current = nexttorun;
  if (unsigned(mem[cpu.pc]) > int(STKMC_nul)) ps = badop;
  else
  { cpu.ir = STKMC_opcodes(mem[cpu.pc]); cpu.pc++;  // fetch
    if (tracing) trace(results, pcnow);
    switch (cpu.ir)                              // execute
    { case STKMC_cal:
        mem[cpu.mp - 2] = process[current].display[mem[cpu.pc]];
                                                 // save display element
        mem[cpu.mp - 3] = cpu.bp;                // save dynamic link
        mem[cpu.mp - 4] = cpu.pc + 2;            // save return address
        process[current].display[mem[cpu.pc]] = cpu.mp;
                                                 // update display
        cpu.bp = cpu.mp;                         // reset base pointer
        cpu.pc = mem[cpu.pc + 1];                // enter procedure
        break;
      case STKMC_ret:
        process[current].display[mem[cpu.pc] - 1] = mem[cpu.bp - 2];
                                                 // restore display
        cpu.sp = cpu.bp - mem[cpu.pc + 1];       // discard stack frame
        cpu.mp = mem[cpu.bp - 5];                // restore mark pointer
        cpu.pc = mem[cpu.bp - 4];                // get return address
        cpu.bp = mem[cpu.bp - 3];                // reset base pointer
        if (cpu.pc == processreturn)             // kill a concurrent process
        { nprocs--; slice = 0;                   // force choice of new process
          if (nprocs == 0)                       // reactivate main program
          { nexttorun = 0; swapregisters(); }
          else                                   // complete this process only
          { chooseprocess();                     // may fail
            process[current].ready = false;
            if (current == nexttorun) ps = deadlock;
          }
        }
        break;
      case STKMC_adr:
        cpu.sp--;
        if (inbounds(cpu.sp))
        { mem[cpu.sp] = process[current].display[mem[cpu.pc] - 1]
                      + mem[cpu.pc + 1];
          cpu.pc += 2;
        }
        break;
      case STKMC_frk:
        nprocs++;
        // first initialize the shadow CPU registers and Display
        process[nprocs].bp = cpu.mp;                    // base pointer
        process[nprocs].mp = cpu.mp;                    // mark pointer
        process[nprocs].sp = cpu.sp;                    // stack pointer
        process[nprocs].pc = mem[cpu.pc];               // process entry point
        process[nprocs].display[0] =
          process[0].display[0];                        // for global access
        process[nprocs].display[1] = cpu.mp;            // for local access
        // now initialize activation record
        mem[process[nprocs].bp - 2] =
          process[0].display[1];                        // display copy
        mem[process[nprocs].bp - 3] = cpu.bp;           // dynamic link
        mem[process[nprocs].bp - 4] = processreturn;    // return address
        // descriptor house keeping
        process[nprocs].stackmax = cpu.mp;              // memory limits
        process[nprocs].stackmin = cpu.mp - partition;
        process[nprocs].ready = true;                   // ready to run
        process[nprocs].queue = 0;                      // clear semaphore queue
        process[nprocs].next = nprocs + 1;              // link to next descriptor
        cpu.sp = cpu.mp - partition;                    // bump parent SP below
        cpu.pc++;                                       // reserved memory
        break;
      case STKMC_cbg:
```

```
  if (mem[cpu.pc] > 0)
  { partition = (cpu.sp - codelen) / mem[cpu.pc]; // divide rest of memory
    parentsp = cpu.sp;                            // for restoration by cnd
  }
  cpu.pc++;
  break;
case STKMC_lit:
  cpu.sp--;
  if (inbounds(cpu.sp)) { mem[cpu.sp] = mem[cpu.pc]; cpu.pc++; }
  break;
case STKMC_dsp:
  cpu.sp -= mem[cpu.pc];
  if (inbounds(cpu.sp)) cpu.pc++;
  break;
case STKMC_brn:
  cpu.pc = mem[cpu.pc]; break;
case STKMC_bze:
  cpu.sp++;
  if (inbounds(cpu.sp))
  { if (mem[cpu.sp - 1] == 0) cpu.pc = mem[cpu.pc]; else cpu.pc++; }
  break;
case STKMC_prs:
  if (tracing) fputs(BLANKS, results);
  loop = mem[cpu.pc];
  cpu.pc++;
  while (inbounds(loop) && mem[loop] != 0)
    { putc(mem[loop], results); loop--; }
  if (tracing) putc('\n', results);
  break;
case STKMC_wgt:
  if (current == 0) ps = badsem;
  else { cpu.sp++; wait(mem[cpu.sp - 1]); }
  break;
case STKMC_sig:
  if (current == 0) ps = badsem;
  else { cpu.sp++; signal(mem[cpu.sp - 1]); }
  break;
case STKMC_cnd:
  if (nprocs > 0)
  { process[nprocs].next = 1;            // close ring
    nexttorun = random(nprocs) + 1;      // choose first process at random
    cpu.sp = parentsp;                   // restore parent stack pointer
  }
  break;
case STKMC_nfn:
  ps = badfun; break;
case STKMC_mst:
  if (inbounds(cpu.sp-STKMC_headersize)) // check space available
  { mem[cpu.sp-5] = cpu.mp;              // save mark pointer
    cpu.mp = cpu.sp;                     // set mark stack pointer
    cpu.sp -= STKMC_headersize;          // bump stack pointer
  }
  break;
case STKMC_add:
  cpu.sp++;
  if (inbounds(cpu.sp)) mem[cpu.sp] += mem[cpu.sp - 1];
  break;
case STKMC_sub:
  cpu.sp++;
  if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
  break;
case STKMC_mul:
  cpu.sp++;
  if (inbounds(cpu.sp)) mem[cpu.sp] *= mem[cpu.sp - 1];
  break;
case STKMC_dvd:
  cpu.sp++;
  if (inbounds(cpu.sp))
  { if (mem[cpu.sp - 1] == 0)
      ps = divzero;
    else
      mem[cpu.sp] /= mem[cpu.sp - 1];
  }
  break;
case STKMC_eql:
  cpu.sp++;
  if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] == mem[cpu.sp - 1]);
  break;
case STKMC_neq:
  cpu.sp++;
  if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] != mem[cpu.sp - 1]);
  break;
case STKMC_lss:
```

```
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] < mem[cpu.sp - 1]);
          break;
        case STKMC_geq:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] >= mem[cpu.sp - 1]);
          break;
        case STKMC_gtr:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] > mem[cpu.sp - 1]);
          break;
        case STKMC_leq:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] <= mem[cpu.sp - 1]);
          break;
        case STKMC_neg:
          if (inbounds(cpu.sp)) mem[cpu.sp] = -mem[cpu.sp];
          break;
        case STKMC_val:
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[cpu.sp] = mem[mem[cpu.sp]];
          break;
        case STKMC_sto:
          cpu.sp++;
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[mem[cpu.sp]] = mem[cpu.sp - 1];
          cpu.sp++;
          break;
        case STKMC_ind:
          if ((mem[cpu.sp + 1] < 0) || (mem[cpu.sp + 1] >= mem[cpu.sp]))
            ps = badind;
          else
          { cpu.sp += 2;
            if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
          }
          break;
        case STKMC_stk:
          stackdump(initsp, results, pcnow); break;
        case STKMC_hlt:
          ps = finished; break;
        case STKMC_inn:
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
          { if (fscanf(data, "%d", &mem[mem[cpu.sp]]) == 0)
              ps = baddata;
            else
              cpu.sp++;
          }
          break;
        case STKMC_prn:
          if (tracing) fputs(BLANKS, results);
          cpu.sp++;
          if (inbounds(cpu.sp)) fprintf(results, " %d", mem[cpu.sp - 1]);
          if (tracing) putc('\n', results);
          break;
        case STKMC_nln:
          putc('\n', results); break;
        case STKMC_nop:
          break;
        default:
          ps = badop; break;
      }
    }
    if (nexttorun != 0) chooseprocess();
  } while (ps == running);
  if (ps != finished) postmortem(results, pcnow);
}

void STKMC::interpret(STKMC_address codelen, STKMC_address initsp)
{ char filename[256];
  FILE *data, *results;
  bool tracing;
  char reply, dummy;
  printf("\nTrace execution (y/N/q)? ");
  reply = getc(stdin); dummy = reply;
  while (dummy != '\n') dummy = getc(stdin);
  if (toupper(reply) != 'Q')
  { tracing = toupper(reply) == 'Y';
    printf("\nData file [STDIN] ? "); gets(filename);
    if (filename[0] == '\0') data = NULL;
    else data = fopen(filename, "r");
    if (data == NULL)
      { printf("taking data from stdin\n"); data = stdin; }
    printf("\nResults file [STDOUT] ? "); gets(filename);
```

```
    if (filename[0] == '\0') results = NULL;
    else results = fopen(filename, "w");
    if (results == NULL)
      { printf("sending results to stdout\n"); results = stdout; }
    emulator(0, codelen, initsp, data, results, tracing);
    if (results != stdout) fclose(results);
    if (data != stdin) fclose(data);
  }
}

STKMC::STKMC()
{ for (int i = 0; i <= STKMC_memsize - 1; i++) mem[i] = 0;
  // Initialize mnemonic table this way for ease of modification in exercises
  mnemonics[STKMC_add] = "ADD";  mnemonics[STKMC_adr] = "ADR";
  mnemonics[STKMC_brn] = "BRN";  mnemonics[STKMC_bze] = "BZE";
  mnemonics[STKMC_cal] = "CAL";  mnemonics[STKMC_cbg] = "CBG";
  mnemonics[STKMC_cnd] = "CND";  mnemonics[STKMC_dsp] = "DSP";
  mnemonics[STKMC_dvd] = "DVD";  mnemonics[STKMC_eql] = "EQL";
  mnemonics[STKMC_frk] = "FRK";  mnemonics[STKMC_geq] = "GEQ";
  mnemonics[STKMC_gtr] = "GTR";  mnemonics[STKMC_hlt] = "HLT";
  mnemonics[STKMC_ind] = "IND";  mnemonics[STKMC_inn] = "INN";
  mnemonics[STKMC_leq] = "LEQ";  mnemonics[STKMC_lit] = "LIT";
  mnemonics[STKMC_lss] = "LSS";  mnemonics[STKMC_mst] = "MST";
  mnemonics[STKMC_mul] = "MUL";  mnemonics[STKMC_neg] = "NEG";
  mnemonics[STKMC_neq] = "NEQ";  mnemonics[STKMC_nfn] = "NFN";
  mnemonics[STKMC_nln] = "NLN";  mnemonics[STKMC_nop] = "NOP";
  mnemonics[STKMC_nul] = "NUL";  mnemonics[STKMC_prn] = "PRN";
  mnemonics[STKMC_prs] = "PRS";  mnemonics[STKMC_ret] = "RET";
  mnemonics[STKMC_sig] = "SIG";  mnemonics[STKMC_stk] = "STK";
  mnemonics[STKMC_sto] = "STO";  mnemonics[STKMC_sub] = "SUB";
  mnemonics[STKMC_val] = "VAL";  mnemonics[STKMC_wgt] = "WGT";
}
```

Appendix C

Cocol grammar for the Clang compiler/interpreter

This appendix gives the Cocol specification and frame file for constructing a compiler for the Clang language as developed by the end of Chapter 18, along with the source for the tree-building code generator.

clang.atg | cgen.h | cgen.cpp | clang.frm

```
----- clang.atg ------------------------------------------------

$CX
COMPILER Clang
/* CLANG level 4 grammar - function, procedures, parameters, concurrency
   Display model.
   Builds an AST for code generation.
   P.D. Terry, Rhodes University, 1996 */

#include "misc.h"
#include "set.h"
#include "table.h"
#include "report.h"
#include "cgen.h"

typedef Set<7> classset;

bool debug;
int blocklevel;
TABLE_idclasses blockclass;

extern TABLE *Table;
extern CGEN *CGen;

/*--------------------------------------------------------------------------*/

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "(*" TO "*)"

CHARACTERS
  cr         = CHR(13) .
  lf         = CHR(10) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  instring   = ANY - "'" - cr - lf .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  string     = "'" (instring | "''") { instring | "''" } "'" .

PRODUCTIONS
  Clang
  =                             (. TABLE_entries entry; TABLE_index index; .)
     "PROGRAM"
     Ident<entry.name>          (. debug = (strcmp(entry.name, "DEBUG") == 0);
                                   entry.idclass = TABLE_progs;
                                   Table->enter(entry, index);
                                   Table->openscope(); .)
     WEAK ";"
     Block<entry.level+1, TABLE_progs, 0>
     "." .

  Block<int blklevel, TABLE_idclasses blkclass, int initialframesize>
  =                             (. int framesize = initialframesize;
                                   CGEN_labels entrypoint;
                                   CGen->jump(entrypoint, CGen->undefined);
                                   if (blklevel > CGEN_levmax) SemError(213); .)
     SYNC
     { (   ConstDeclarations
         | VarDeclarations<framesize>
```

```
      | ProcDeclaration
    ) SYNC }                (. /* blockclass, blocklevel global for efficiency */
                               blockclass = blkclass; blocklevel = blklevel;
                               CGen->backpatch(entrypoint);
                               /* reserve space for variables */
                               CGen->openstackframe(framesize
                                                    - initialframesize); .)
   CompoundStatement        (. switch (blockclass)
                               { case TABLE_progs :
                                   CGen->leaveprogram(); break;
                                 case TABLE_procs :
                                   CGen->leaveprocedure(blocklevel); break;
                                 case TABLE_funcs :
                                   CGen->functioncheck(); break;
                               }
                               if (debug) /* demonstration purposes */
                                 Table->printtable(stdout);
                               Table->closescope(); .) .

ConstDeclarations
=  "CONST" OneConst { OneConst } .

OneConst
=                           (. TABLE_entries entry; TABLE_index index; .)
   Ident<entry.name>        (. entry.idclass = TABLE_consts; .)
   WEAK "="
   Number<entry.c.value>    (. Table->enter(entry, index) .)
   ";" .

VarDeclarations<int &framesize>
=  "VAR" OneVar<framesize> { WEAK "," OneVar<framesize> } ";" .

OneVar<int &framesize>
=                           (. TABLE_entries entry; TABLE_index index;
                               entry.idclass = TABLE_vars; entry.v.ref = false;
                               entry.v.size = 1; entry.v.scalar = true;
                               entry.v.offset = framesize + 1; .)
   Ident<entry.name>
   [ UpperBound<entry.v.size> (. entry.v.scalar = false; .)
   ]                        (. Table->enter(entry, index);
                               framesize += entry.v.size; .) .

UpperBound<int &size>
=  "[" Number<size> "]"     (. size++; .) .

ProcDeclaration
=                           (. TABLE_entries entry; TABLE_index index; .)
   (   "PROCEDURE"          (. entry.idclass = TABLE_procs; .)
     | "FUNCTION"           (. entry.idclass = TABLE_funcs; .)
   ) Ident<entry.name>      (. entry.p.params = 0; entry.p.paramsize = 0;
                               entry.p.firstparam = NULL;
                               CGen->storelabel(entry.p.entrypoint);
                               Table->enter(entry, index);
                               Table->openscope() .)
   [
   FormalParameters<entry>  (. Table->update(entry, index) .)
   ] WEAK ";"
   Block<entry.level+1, entry.idclass, entry.p.paramsize + CGEN_headersize>
   ";" .

FormalParameters<TABLE_entries &proc>
=                           (. TABLE_index p; .)
   "(" OneFormal<proc, proc.p.firstparam>
   { WEAK "," OneFormal<proc, p> } ")" .

OneFormal<TABLE_entries &proc, TABLE_index &index>
=                           (. TABLE_entries formal;
                               formal.idclass = TABLE_vars; formal.v.ref = false;
                               formal.v.size = 1; formal.v.scalar = true;
                               formal.v.offset = proc.p.paramsize
                                                 + CGEN_headersize + 1 .)
   Ident<formal.name>
   [ "[" "]"                (. formal.v.size = 2; formal.v.scalar = false;
                               formal.v.ref = true; .)
   ]                        (. Table->enter(formal, index);
                               proc.p.paramsize += formal.v.size;
                               proc.p.params++; .) .

CompoundStatement
=  "BEGIN" Statement { WEAK ";" Statement } "END" .

Statement
=  SYNC [  CompoundStatement | AssignmentOrCall | ReturnStatement
```

```
          | IfStatement      | WhileStatement
          | CobeginStatement | SemaphoreStatement
          | ReadStatement    | WriteStatement
          | "STACKDUMP"       (. CGen->dump(); .)
        ] .

AssignmentOrCall
=                           (. TABLE_entries entry;
                               AST des, exp;.)
   Designator<des, classset(TABLE_vars, TABLE_procs), entry, true>
   (   /* assignment */     (. if (entry.idclass != TABLE_vars) SemError(210); .)
      ":=" Expression<exp, true>
      SYNC                  (. CGen->assign(des, exp); .)
     | /* procedure call */ (. if (entry.idclass < TABLE_procs)
                               { SemError(210); return; }
                               CGen->markstack(des, entry.level,
                                               entry.p.entrypoint); .)
      ActualParameters<des, entry>
                           (. CGen->call(des); .)
   ) .

Designator<AST &D, classset allowed, TABLE_entries &entry, bool entire>
=                           (. TABLE_alfa name;
                               AST index, size;
                               bool found;
                               D = CGen->emptyast(); .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (!allowed.memb(entry.idclass)) SemError(206);
                               if (entry.idclass != TABLE_vars) return;
                               CGen->stackaddress(D, entry.level, entry.v.offset,
                                                  entry.v.ref); .)
   (   "["                  (. if (entry.v.scalar) SemError(204); .)
       Expression<index, true>
                           (. if (!entry.v.scalar)
                               /* determine size for bounds check */
                               { if (entry.v.ref)
                                   CGen->stackaddress(size, entry.level,
                                                      entry.v.offset + 1, false);
                                 else
                                   CGen->stackconstant(size, entry.v.size);
                                 CGen->subscript(D, entry.v.ref, entry.level,
                                                 entry.v.offset, size, index);
                               } .)
       "]"
     |                      (. if (!entry.v.scalar)
                               { if (entire) SemError(205);
                                 if (entry.v.ref)
                                   CGen->stackaddress(size, entry.level,
                                                      entry.v.offset + 1, false);
                                 else
                                   CGen->stackconstant(size, entry.v.size);
                                 CGen->stackreference(D, entry.v.ref, entry.level,
                                                      entry.v.offset, size);
                               } .)
   ) .

ActualParameters<AST &p, TABLE_entries proc>
=                           (. int actual = 0; .)
   [  "("                   (. actual++; .)
      OneActual<p, (*Table).isrefparam(proc, actual)>
      { WEAK "," }          (. actual++; .)
      OneActual<p, (*Table).isrefparam(proc, actual)> } ")"
   ]                        (. if (actual != proc.p.params) SemError(209); .) .

OneActual<AST &p, bool byref>
=                           (. AST par; .)
   Expression<par, !byref>  (. if (byref && !CGen->isrefast(par)) SemError(214);
                               CGen->linkparameter(p, par); .) .

ReturnStatement
=                           (. AST dest, exp; .)
   "RETURN"
   (                        (. if (blockclass != TABLE_funcs) SemError(219);
                               CGen->stackaddress(dest, blocklevel, 1, false); .)
       Expression<exp, true>
                           (. CGen->assign(dest, exp);
                               CGen->leavefunction(blocklevel); .)
     | /* empty */          (. switch (blockclass)
                               { case TABLE_procs :
                                   CGen->leaveprocedure(blocklevel); break;
                                 case TABLE_progs :
                                   CGen->leaveprogram(); break;
```

```
                                 case TABLE_funcs : SemError(220); break;
                               } .)
   ) .

IfStatement
=                          (. CGEN_labels testlabel;
                              AST C; .)
   "IF" Condition<C> "THEN" (. CGen->jumponfalse(C, testlabel, CGen->undefined) .)
   Statement               (. CGen->backpatch(testlabel); .) .

WhileStatement
=                          (. CGEN_labels startloop, testlabel, dummylabel;
                              AST  C; .)
   "WHILE"                 (. CGen->storelabel(startloop) .)
   Condition<C> "DO"       (. CGen->jumponfalse(C, testlabel, CGen->undefined) .)
   Statement               (. CGen->jump(dummylabel, startloop);
                              CGen->backpatch(testlabel) .) .

Condition<AST &C>
=                          (. AST E;
                              CGEN_operators op; .)
   Expression<C, true>
   (  RelOp<op>
      Expression<E, true>  (. CGen->comparison(op, C, E); .)
    | /* Missing op */     (. SynError(91) .)
   ) .

CobeginStatement
=                          (. int processes = 0;
                              CGEN_labels start; .)
   "COBEGIN"               (. if (blockclass != TABLE_progs) SemError(215);
                              CGen->cobegin(start); .)
   ProcessCall             (. processes++; .)
   { WEAK ";" ProcessCall  (. processes++; .)
   }
   "COEND"                 (. CGen->coend(start, processes); .) .

ProcessCall
=                          (. TABLE_entries entry;
                              AST P; .)
   Designator<P, classset(TABLE_procs), entry, true>
                           (. if (entry.idclass < TABLE_procs) return;
                              CGen->markstack(P, entry.level,
                                              entry.p.entrypoint); .)
   ActualParameters<P, entry>
                           (. CGen->forkprocess(P); .) .

SemaphoreStatement
=                          (. bool wait;
                              AST sem; .)
   (   "WAIT"              (. wait = true; .)
     | "SIGNAL"            (. wait = false; .)
   )
   "(" Variable<sem>       (. if (wait) CGen->waitop(sem);
                              else CGen->signalop(sem); .)
   ")" .

ReadStatement
=                          (. AST V; .)
   "READ" "(" Variable<V>  (. CGen->readvalue(V); .)
   { WEAK "," Variable<V>  (. CGen->readvalue(V); .)
   } ")" .

Variable<AST &V>
=                          (. TABLE_entries entry; .)
   Designator<V, classset(TABLE_vars), entry, true> .

WriteStatement
=  "WRITE" [ "(" WriteElement { WEAK "," WriteElement }  ")" ]
                           (. CGen->newline(); .) .

WriteElement
=                          (. AST exp;
                              char str[600];
                              CGEN_labels startstring; .)
    String<str>            (. CGen->stackstring(str, startstring);
                              CGen->writestring(startstring); .)
  | Expression<exp, true>  (. CGen->writevalue(exp) .) .

Expression<AST &E, bool entire>
=                          (. AST T;
                              CGEN_operators op;
                              E = CGen->emptyast(); .)
```

```
   (   "+" Term<E, true>
     | "-" Term<E, true>    (. CGen->negateinteger(E); .)
     | Term<E, entire>
   )
   { AddOp<op> Term<T, true>(. CGen->binaryintegerop(op, E, T); .)
   } .

Term<AST &T, bool entire>
=                          (. AST F;
                              CGEN_operators op; .)
   Factor<T, entire>
   { (  MulOp<op>
     | /* missing op */    (. SynError(92); op = CGEN_opmul; .)
     ) Factor<F, true>     (. CGen->binaryintegerop(op, T, F); .)
   } .

Factor<AST &F, bool entire>
=                          (. TABLE_entries entry;
                              int value;
                              F = CGen->emptyast(); .)
   Designator<F, classset(TABLE_consts, TABLE_vars, TABLE_funcs), entry, entire>
                           (. switch (entry.idclass)
                              { case TABLE_consts :
                                  CGen->stackconstant(F, entry.c.value);
                                  return;
                                case TABLE_procs :
                                case TABLE_funcs :
                                  CGen->markstack(F, entry.level,
                                                  entry.p.entrypoint); break;
                                case TABLE_vars :
                                case TABLE_progs :
                                  return;
                              } .)
   ActualParameters<F, entry>
 | Number<value>           (. CGen->stackconstant(F, value) .)
 | "(" Expression<F, true> ")" .

AddOp<CGEN_operators &op>
=    "+"                   (. op = CGEN_opadd; .)
   | "-"                   (. op = CGEN_opsub; .) .

MulOp<CGEN_operators &op>
=    "*"                   (. op = CGEN_opmul; .)
   | "/"                   (. op = CGEN_opdvd; .) .

RelOp<CGEN_operators &op>
=    "="                   (. op = CGEN_opeql; .)
   | "<>"                  (. op = CGEN_opneq; .)
   | "<"                   (. op = CGEN_oplss; .)
   | "<="                  (. op = CGEN_opleq; .)
   | ">"                   (. op = CGEN_opgtr; .)
   | ">="                  (. op = CGEN_opgeq; .) .

Ident<char *name>
=  identifier              (. LexName(name, TABLE_alfalength); .) .

String<char *str>
=  string                  (. char local[100];
                              LexString(local, sizeof(local) - 1);
                              int i = 0;
                              while (local[i]) /* strip quotes */
                              { local[i] = local[i+1]; i++; }
                              local[i-2] = '\0';
                              i = 0;
                              while (local[i]) /* find internal quotes */
                              { if (local[i] == '\'')
                                  { int j = i;
                                    while (local[j])
                                    { local[j] = local[j+1]; j++; }
                                  }
                                  i++;
                              }
                              strcpy(str, local); .) .

Number <int &num>
=  number                  (. char str[100];
                              int i = 0, l, digit, overflow = 0;
                              num = 0;
                              LexString(str, sizeof(str) - 1);
                              l = strlen(str);
                              while (i <= l && isdigit(str[i])) {
                                digit = str[i] - '0'; i++;
                                if (num <= (maxint - digit) / 10)
```

```
                                    num = 10 * num + digit;
                                  else overflow = 1;
                                }
                                if (overflow) SemError(200); .) .

END Clang.


----- cgen.h ---------------------------------------------------

// Code Generation for Clang level 4 compiler/interpreter
// AST version for stack machine (OOP)
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#ifndef CGEN_H
#define CGEN_H

#include "misc.h"
#include "stkmc.h"
#include "report.h"

#define CGEN_headersize  STKMC_headersize
#define CGEN_levmax      STKMC_levmax

enum CGEN_operators {
  CGEN_opadd, CGEN_opsub, CGEN_opmul, CGEN_opdvd, CGEN_opeql, CGEN_opneq,
  CGEN_oplss, CGEN_opgeq, CGEN_opgtr, CGEN_opleq
};

struct NODE;
typedef NODE *AST;
typedef short CGEN_labels;

class CGEN {
  public:
    CGEN_labels undefined;    // for forward references

    CGEN(REPORT *R);
    // Initializes code generator

    AST emptyast(void);
    // Returns an empty (undefined) AST

    bool isrefast(AST a);
    // Returns true if a corresponds to a reference AST

    void negateinteger(AST &i);
    // Generates code to negate integer i

    void binaryintegerop(CGEN_operators op, AST &l, AST &r);
    // Generates code to perform infix operation op on l, r

    void comparison(CGEN_operators op, AST &l, AST &r);
    // Generates code to perform comparison operation op on l, r

    void readvalue(AST i);
    // Generates code to read value for i

    void writevalue(AST i);
    // Generates code to output value i

    void newline(void);
    // Generates code to output line mark

    void writestring(CGEN_labels location);
    // Generates code to output string stored at known location

    void stackstring(char *str, CGEN_labels &location);
    // Stores str in literal pool in memory and return its location

    void stackconstant(AST &c, int number);
    // Creates constant AST for constant c from number

    void stackaddress(AST &v, int level, int offset, bool byref);
    // Creates address AST for variable v with known level, offset

    void dereference(AST &a);
    // Generates code to replace address a by the value stored there

    void stackreference(AST &base, bool byref,
                        int level, int offset, AST &size);
```

```
    // Creates an actual parameter node for a reference parameter corresponding
    // to an array with given base and size

    void subscript(AST &base, bool byref,
                   int level, int offset, AST &size, AST &index);
    // Prepares to apply an index to an array with given base, with checks
    // that the limit on the bounds is not exceeded

    void assign(AST dest, AST expr);
    // Generates code to store value of expr on dest

    void openstackframe(int size);
    // Generates code to reserve space for size variables

    void leaveprogram(void);
    // Generates code needed to leave a program (halt)

    void leaveprocedure(int blocklevel);
    // Generates code needed to leave a regular procedure at a given blocklevel

    void leavefunction(int blocklevel);
    // Generates code needed to leave a function at given blockLevel

    void functioncheck(void);
    // Generates code to ensure that a function has returned a value

    void storelabel(CGEN_labels &location);
    // Stores address of next instruction in location for use in backpatching

    void jump(CGEN_labels &here, CGEN_labels destination);
    // Generates unconditional branch from here to destination

    void jumponfalse(AST condition, CGEN_labels &here, CGEN_labels destination);
    // Generates branch from here to destination, dependent on condition

    void backpatch(CGEN_labels location);
    // Stores the current location counter as the address field of the branch
    // instruction currently held in an incomplete form at location

    void markstack(AST &p, int level, int entrypoint);
    // Generates code to reserve mark stack storage before calling procedure p
    // with known level and entrypoint

    void linkparameter(AST &p, AST &par);
    // Adds par to the actual parameter list for call to procedure p

    void call(AST &p);
    // Generates code to enter procedure p

    void cobegin(CGEN_labels &location);
    // Generates code to initiate concurrent processing

    void coend(CGEN_labels location, int number);
    // Generates code to terminate concurrent processing

    void forkprocess(AST &p);
    // Generates code to initiate process p

    void signalop(AST s);
    // Generates code for semaphore signalling operation on s

    void waitop(AST s);
    // Generates code for semaphore wait operation on s

    void dump(void);
    // Generates code to dump the current state of the evaluation stack

    void getsize(int &codelength, int &initsp);
    // Returns length of generated code and initial stack pointer

    int gettop(void);
    // Returns codetop

    void emit(int word);
    // Emits single word

  private:
    REPORT *Report;
    bool generatingcode;
    STKMC_address codetop, stktop;
    void binaryop(CGEN_operators op, AST &left, AST &right);
};
```

```
#endif /*CGEN_H*/


----- cgen.cpp -------------------------------------------------

// Code Generation for Clang Level 4 compiler/interpreter
// AST version for stack machine (OOP)
// Includes procedures, functions, parameters, arrays, concurrency.
// Display machine.
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "cgen.h"
#include "report.h"

extern STKMC  *Machine;
extern CGEN   *CGen;
extern REPORT *Report;

// ++++++++++++++ AST node classes ++++++++++++++++++++++++++++++++++++++++

struct NODE {
  int value;       // value derived from this node
  bool defined;    // true if value is defined (for constant nodes)
  bool refnode;    // true if node corresponds to a reference
  NODE()                          { defined = false; refnode = false; }
  virtual void emit1(void)    = 0;
  virtual void emit2(void)    = 0;
  virtual void link(AST next) = 0;
};

struct BINOPNODE : public NODE {
  CGEN_operators op;
  AST left, right;
  BINOPNODE(CGEN_operators O, AST L, AST R) { op = O; left = L; right = R; }
  virtual void emit1(void);      // load value onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST next)    {;}
};

void BINOPNODE::emit1(void)
// load value onto stack resulting from binary operation
{ bool folded = false;
  if (left && right)
  { // Some optimizations (others are left as an exercise).
    // These need improvement so as to perform range checking
    switch (op)
    { case CGEN_opadd:
        if (right->defined && right->value == 0) // x + 0 = x
          { left->emit1(); folded = true; }
        else if (left->defined && left->value == 0) // 0 + x = x
          { right->emit1(); folded = true; }
        break;

      case CGEN_opsub:
        if (right->defined && right->value == 0) // x - 0 = x
          { left->emit1(); folded = true; }
        else if (left->defined && left->value == 0) // 0 - x = -x
          { right->emit1(); CGen->emit(int(STKMC_neg)); folded = true; }
        break;

      case CGEN_opmul:
        if (right->defined && right->value == 1) // x * 1 = x
          { left->emit1(); folded = true; }
        else if (left->defined && left->value == 1) // 1 * x = x
          { right->emit1(); folded = true; }
        else if (right->defined && right->value == 0) // x * 0 = 0
          { right->emit1(); folded = true; }
        else if (left->defined && left->value == 0) // 0 * x = 0
          { left->emit1(); folded = true; }
        break;

      case CGEN_opdvd:
        if (right->defined && right->value == 1) // x / 1 = x
          { left->emit1(); folded = true; }
        else if (right->defined && right->value == 0) // x / 0 = error
          { Report->error(224); folded = true; }
        break;
      // no folding attempted here for relational operations
    }
  }
  if (!folded)
  { if (left) left->emit1(); if (right) right->emit1();
```

```
    CGen->emit(int(STKMC_add) + int(op));   // careful - ordering used
  }
  if (left) delete left; if (right) delete right;
}

struct MONOPNODE : public NODE {
  CGEN_operators op;  // for expansion - only negation used here
  AST operand;
  MONOPNODE(CGEN_operators O, AST E) { op = O; operand = E; }
  virtual void emit1(void);      // load value onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST next)    {;}
};

void MONOPNODE::emit1(void)
// load value onto stack resulting from unary operation
{ if (operand) { operand->emit1(); delete operand; }
  CGen->emit(int(STKMC_neg));
}

struct VARNODE : public NODE {
  bool ref;       // direct or indirectly accessed
  int level;      // static level of declaration
  int offset;     // offset of variable assigned by compiler
  VARNODE() {;}   // default
  VARNODE(bool byref, int L, int O)
    { ref = byref; level = L; offset = O; }
  virtual void emit1(void);      // load variable value onto stack
  virtual void emit2(void);      // load variable address onto stack
  virtual void link(AST next)    {;}
};

void VARNODE::emit1(void)
// load variable value onto stack
{ emit2(); CGen->emit(int(STKMC_val)); }

void VARNODE::emit2(void)
// load variable address onto stack
{ CGen->emit(int(STKMC_adr)); CGen->emit(level); CGen->emit(-offset);
  if (ref) CGen->emit(int(STKMC_val));
}

struct INDEXNODE : public VARNODE {
  AST size;      // for range checking
  AST index;     // subscripting expression
  INDEXNODE(bool byref, int L, int O, AST S, AST I)
    { ref = byref; level = L; offset = O; size = S; index = I; }
  virtual void emit2(void);      // load array element address and check
  virtual void link(AST next)    {;}
};

void INDEXNODE::emit2(void)
// load array element address and check in range
{ CGen->emit(int(STKMC_adr)); CGen->emit(level); CGen->emit(-offset);
  if (ref) CGen->emit(int(STKMC_val));
  if (index) { index->emit1(); delete index; }
  if (size) { size->emit1(); delete size; }
  CGen->emit(int(STKMC_ind));
}

// void INDEXNODE::emit1(void) is inherited from VARNODE

struct REFNODE : public VARNODE {
  AST size;
  REFNODE(bool byref, int L, int O, AST S)
    { ref = byref; level = L; offset = O; size = S; refnode = 1; }
  virtual void emit1(void);       // load array argument address and size
  virtual void emit2(void)        {;}
  virtual void link(AST next)     {;}
};

void REFNODE::emit1(void)
// load array argument address and size
{ CGen->emit(int(STKMC_adr)); CGen->emit(level); CGen->emit(-offset);
  if (ref) CGen->emit(int(STKMC_val));
  if (size) { size->emit1(); delete size; }
}

struct CONSTNODE : public NODE {
  CONSTNODE(int V)               { value = V; defined = true; }
  virtual void emit1(void);      // load constant value onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST next)    {;}
```

```
};

void CONSTNODE::emit1(void)
// load constant value onto stack
{ CGen->emit(int(STKMC_lit)); CGen->emit(value); }

struct PARAMNODE : public NODE {
  AST par, next;
  PARAMNODE(AST P)               { par = P; next = NULL; }
  virtual void emit1(void);      // load actual parameter onto stack
  virtual void emit2(void)       {;}
  virtual void link(AST param)   { next = param; }
};

void PARAMNODE::emit1(void)
// load actual parameter onto stack
{ if (par) { par->emit1(); delete par; }
  if (next) { next->emit1(); delete next; }  // follow link to next parameter
}

struct PROCNODE : public NODE {
  int level, entrypoint;     // static level and first instruction
  AST firstparam, lastparam; // pointers to argument list
  PROCNODE(int L, int ent)
    { level = L; entrypoint = ent; firstparam = NULL; lastparam = NULL; }
  virtual void emit1(void);      // generate procedure/function call
  virtual void emit2(void);      // generate process call
  virtual void link(AST next);   // link next actual parameter
};

void PROCNODE::emit1(void)
// generate procedure/function call
{ CGen->emit(int(STKMC_mst));
  if (firstparam) { firstparam->emit1(); delete firstparam; }
  CGen->emit(int(STKMC_cal));
  CGen->emit(level);
  CGen->emit(entrypoint);
}

void PROCNODE::emit2(void)
// generate process call
{ CGen->emit(int(STKMC_mst));
  if (firstparam) { firstparam->emit1(); delete firstparam; }
  CGen->emit(int(STKMC_frk));
  CGen->emit(entrypoint);
}

void PROCNODE::link(AST param)
// link next actual parameter
{ if (!firstparam) firstparam = param; else lastparam->link(param);
  lastparam = param;
}

// +++++++++++++++ code generator constructor ++++++++++++++++++++++++++

CGEN::CGEN(REPORT *R)
{ undefined = 0;    // for forward references (exported)
  Report = R;
  generatingcode = true;
  codetop = 0;
  stktop = STKMC_memsize - 1;
}

void CGEN::emit(int word)
// Code generator for single word
{ if (!generatingcode) return;
  if (codetop >= stktop) { Report->error(212); generatingcode = false; }
  else { Machine->mem[codetop] = word; codetop++; }
}

bool CGEN::isrefast(AST a)
{ return a && a->refnode; }

// +++++++++++++++ routines that build the tree ++++++++++++++++++++++++++

AST CGEN::emptyast(void)
{ return NULL; }

void CGEN::negateinteger(AST &i)
{ if (i && i->defined) { i->value = -i->value; return; }  // simple folding
  i = new MONOPNODE(CGEN_opsub, i);
}
```

```
void CGEN::binaryop(CGEN_operators op, AST &left, AST &right)
{ if (left && right && left->defined && right->defined)
  { // simple constant folding - better range checking needed
    switch (op)
    { case CGEN_opadd: left->value += right->value; break;
      case CGEN_opsub: left->value -= right->value; break;
      case CGEN_opmul: left->value *= right->value; break;
      case CGEN_opdvd:
        if (right->value == 0) Report->error(224);
        else left->value /= right->value;
        break;
      case CGEN_oplss: left->value = (left->value < right->value); break;
      case CGEN_opgtr: left->value = (left->value > right->value); break;
      case CGEN_opleq: left->value = (left->value <= right->value); break;
      case CGEN_opgeq: left->value = (left->value >= right->value); break;
      case CGEN_opeql: left->value = (left->value == right->value); break;
      case CGEN_opneq: left->value = (left->value != right->value); break;
    }
    delete right;
    return;
  }
  left = new BINOPNODE(op, left, right);
}

void CGEN::binaryintegerop(CGEN_operators op, AST &l, AST &r)
{ binaryop(op, l, r); }

void CGEN::comparison(CGEN_operators op, AST &l, AST &r)
{ binaryop(op, l, r); }

void CGEN::stackconstant(AST &c, int number)
{ c = new CONSTNODE(number); }

void CGEN::stackaddress(AST &v, int level, int offset, bool byref)
{ v = new VARNODE(byref, level, offset); }

void CGEN::linkparameter(AST &p, AST &par)
{ AST param = new PARAMNODE(par); p->link(param); }

void CGEN::stackreference(AST &base, bool byref, int level, int offset,
                          AST &size)
{ if (base) delete base; base = new REFNODE(byref, level, offset, size); }

void CGEN::subscript(AST &base, bool byref, int level, int offset,
                     AST &size, AST &index)
// Note the folding of constant indexing of arrays, and compile time
// range checking
{ if (!index || !index->defined || !size || !size->defined)
  { if (base) delete base;
    base = new INDEXNODE(byref, level, offset, size, index);
    return;
  }
  if (unsigned(index->value) >= size->value) // range check immediately
    Report->error(223);
  else
  { if (base) delete base;
    base = new VARNODE(byref, level, offset + index->value);
  }
  delete index; delete size;
}

void CGEN::markstack(AST &p, int level, int entrypoint)
{ p = new PROCNODE(level, entrypoint); }

// +++++++++++++++ code generation requiring tree walk ++++++++++++++++++

void CGEN::jumponfalse(AST condition, CGEN_labels &here,
                       CGEN_labels destination)
{ if (condition) { condition->emit1(); delete condition; }
  here = codetop; emit(int(STKMC_bze)); emit(destination);
}

void CGEN::assign(AST dest, AST expr)
{ if (dest) { dest->emit2(); delete dest; }
  if (expr) { expr->emit1(); delete expr; emit(int(STKMC_sto)); }
}

void CGEN::readvalue(AST i)
{ if (i) { i->emit2(); delete i; } emit(int(STKMC_inn)); }

void CGEN::writevalue(AST i)
{ if (i) { i->emit1(); delete i; } emit(int(STKMC_prn)); }
```

```
void CGEN::call(AST &p)
{ if (p) { p->emit1(); delete p; } }

void CGEN::signalop(AST s)
{ if (s) { s->emit2(); delete s; } emit(int(STKMC_sig)); }

void CGEN::waitop(AST s)
{ if (s) { s->emit2(); delete s; } emit(int(STKMC_wgt)); }

void CGEN::forkprocess(AST &p)
{ if (p) { p->emit2(); delete p; } }

// +++++++++++++++ code generation not requiring tree walk ++++++++++++++

void CGEN::newline(void)
{ emit(int(STKMC_nln)); }

void CGEN::writestring(CGEN_labels location)
{ emit(int(STKMC_prs)); emit(location); }

void CGEN::stackstring(char *str, CGEN_labels &location)
{ int l = strlen(str);
  if (stktop <= codetop + l + 1)
    { Report->error(212); generatingcode = false; return; }
  location = stktop - 1;
  for (int i = 0; i < l; i++) { stktop--; Machine->mem[stktop] = str[i]; }
  stktop--; Machine->mem[stktop] = 0;
}

void CGEN::dereference(AST &a)
{ /* not needed */ }

void CGEN::openstackframe(int size)
{ if (size > 0) { emit(int(STKMC_dsp)); emit(size); } }

void CGEN::leaveprogram(void)
{ emit(int(STKMC_hlt)); }

void CGEN::leavefunction(int blocklevel)
{ emit(int(STKMC_ret)); emit(blocklevel); emit(1); }

void CGEN::functioncheck(void)
{ emit(int(STKMC_nfn)); }

void CGEN::leaveprocedure(int blocklevel)
{ emit(int(STKMC_ret)); emit(blocklevel); emit(0); }

void CGEN::cobegin(CGEN_labels &location)
{ location = codetop; emit(int(STKMC_cbg)); emit(undefined); }

void CGEN::coend(CGEN_labels location, int number)
{ if (number >= STKMC_procmax) Report->error(216);
  else { Machine->mem[location+1] = number; emit(int(STKMC_cnd)); }
}

void CGEN::storelabel(CGEN_labels &location)
{ location = codetop; }

void CGEN::jump(CGEN_labels &here, CGEN_labels destination)
{ here = codetop; emit(int(STKMC_brn)); emit(destination); }

void CGEN::backpatch(CGEN_labels location)
{ if (codetop == location + 2 &&
      STKMC_opcodes(Machine->mem[location]) == STKMC_brn)
    codetop -= 2;
  else
    Machine->mem[location+1] = codetop;
}

void CGEN::dump(void)
{ emit(int(STKMC_stk)); }

void CGEN::getsize(int &codelength, int &initsp)
{ codelength = codetop; initsp = stktop; }

int CGEN::gettop(void) { return codetop; }


----- clang.frm -----------------------------------------------------

/* Clang compiler generated by Coco/R 1.06 (C++ version) */

#include  <stdio.h>
```

```
#include  <fcntl.h>
#include  <stdlib.h>
#include  <string.h>

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  include  <io.h>
#else
#  include <unistd.h>
#  define  O_BINARY     0
#endif

#include "misc.h"
#include "set.h"
#include "table.h"
#include "cgen.h"
#include "stkmc.h"

typedef Set<7> classset;

#include -->ScanHeader
#include -->ParserHeader
#include "cr_error.hpp"

static FILE *lst;
static char SourceName[256], ListName[256], CodeName[256];

TABLE  *Table;
CGEN   *CGen;
STKMC  *Machine;
REPORT *Report;

class clangError : public CRError {
  public:
    clangError(char *name, AbsScanner *S) : CRError(name, S, MAXERROR) {};
    virtual char *GetUserErrorMsg(int n);
    virtual char *GetErrorMsg(int n)
      { if (n <= MAXERROR) return ErrorMsg[n]; else return GetUserErrorMsg(n); }
  private:
    static char *ErrorMsg[];
};

char *clangError::ErrorMsg[] = {
#include -->ErrorHeader
"User error number clash",
""
};

char *clangError::GetUserErrorMsg(int n)
{ switch (n) {
  // first few are extra syntax errors
    case 91:  return "Relational operator expected";
    case 92:  return "Malformed expression";
    case 93:  return "Bad declaration order";
  // remainder are constraint (static semantic) errors
    case 200: return "Constant out of range";
    case 201: return "Identifier redeclared";
    case 202: return "Undeclared identifier";
    case 203: return "Unexpected parameters";
    case 204: return "Unexpected subscript";
    case 205: return "Subscript required";
    case 206: return "Invalid class of identifier";
    case 207: return "Variable expected";
    case 208: return "Too many formal parameters";
    case 209: return "Wrong number of parameters";
    case 210: return "Invalid assignment";
    case 211: return "Cannot read this type of variable";
    case 212: return "Program too long";
    case 213: return "Too deeply nested";
    case 214: return "Invalid parameter";
    case 215: return "COBEGIN only allowed in main program";
    case 216: return "Too many concurrent processes";
    case 217: return "Only global procedure calls allowed here";
    case 218: return "Type mismatch";
    case 219: return "Unexpected expression";
    case 220: return "Missing expression";
    case 221: return "Boolean expression required";
    case 222: return "Invalid expression";
    case 223: return "Index out of range";
    case 224: return "Division by zero";
    default:  return "Compiler error";
  }
}
```

```
class clangReport : public REPORT {
// interface for code generators and other auxiliaries
  public:
    clangReport(clangError *E)
      { Error = E; }
    virtual void error(int errorcode)
      { Error->ReportError(errorcode); errors = true; }
  private:
    clangError *Error;
};

void main(int argc, char *argv[])
{ int codelength, initsp;
  int S_src;
  char reply;
  lst = stderr;

  // check on correct parameter usage
  if (argc < 2) { fprintf(stderr, "No input file specified\n"); exit(1); }

  // open the source file
  strcpy(SourceName, argv[1]);
  if ((S_src = open(SourceName, O_RDONLY | O_BINARY)) == -1)
  { fprintf(stderr, "Unable to open input file %s\n", SourceName); exit(1); }

  if (argc > 2) strcpy(ListName, argv[2]);
    else appendextension(SourceName, ".lst", ListName);
  if ((lst = fopen(ListName, "w")) == NULL)
  { fprintf(stderr, "Unable to open list file %s\n", ListName); exit(1); }

  // instantiate Scanner, Parser and Error handlers

  -->ScanClass *Scanner = new -->ScanClass(S_src, -->IgnoreCase);
  clangError   *Error = new clangError(SourceName, Scanner);
  -->ParserClass  *Parser = new -->ParserClass(Scanner, Error);
  Report  = new clangReport(Error);
  CGen    = new CGEN(Report);
  Table   = new TABLE(Report);
  Machine = new STKMC();

  // parse the source
  Parser->Parse();
  close(S_src);

  // generate source listing
  Error->SetOutput(lst);
  Error->PrintListing(Scanner);
  fclose(lst);

  // list generated code for interest
  CGen->getsize(codelength, initsp);
  appendextension(SourceName, ".cod", CodeName);
  Machine->listcode(CodeName, codelength);

  if (Error->Errors)
    fprintf(stderr, "Compilation failed - see %s\n", ListName);
  else
  { fprintf(stderr, "Compilation successful\n");
    while (true)
     { printf("\nInterpret? (y/n) ");
       do
       { scanf("%c", &reply);
       } while (toupper(reply) != 'N' && toupper(reply) != 'Y');
       if (toupper(reply) == 'N') break;
       scanf("%*[^\n]"); getchar();
       Machine->interpret(codelength, initsp);
     }
  }

  delete Scanner;
  delete Parser;
  delete Error;
  delete Table;
  delete Report;
  delete CGen;
  delete Machine;
}
```

Appendix D

Source code for a macro assembler

This appendix gives the complete source code for the macro assembler for the single-accumulator machine discussed in Chapter 7.

assemble.cpp | misc.h | set.h | sh.s | sh.cpp | la.h | la.cpp | sa.h | sa.cpp | st.h | st.cpp | st.h | st.cpp | mh.h | mh.cpp | asmbase.h | as.h | as.cpp | mc.h | mc.cpp

```
----- assemble.cpp ------------------------------------------------------------

// Macro assembler/interpreter for the single-accumulator machine
// P.D. Terry,  Rhodes University, 1996

#include "mc.h"
#include "as.h"

#define version         "Macro Assembler 1.0"
#define usage           "Usage: ASSEMBLE source [listing]\n"

void main(int argc, char *argv[])
{ bool errors;
  char reply;
  char sourcename[256], listname[256];

  // check on correct parameter usage
  if (argc == 1) { printf(usage); exit(1); }
  strcpy(sourcename, argv[1]);
  if (argc > 2) strcpy(listname, argv[2]);
  else appendextension(sourcename, ".lst", listname);

  MC *Machine   = new(MC);
  AS *Assembler = new AS(sourcename, listname, version, Machine);
  Assembler->assemble(errors);
  if (errors)
  { printf("\nAssembly failed\n"); }
  else
  { printf("\nAssembly successful\n");
    while (true)
    { printf("\nInterpret? (y/n) ");
      do
      { scanf("%c", &reply);
      } while (toupper(reply) != 'N' && toupper(reply) != 'Y');
      if (toupper(reply) == 'N') break;
      scanf("%*[^\n]"); getchar();
      Machine->interpret();
    }
  }
  delete Machine;
  delete Assembler;
}


----- misc.h -----------------------------------------------------------------

// Various common items for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef MISC_H
#define MISC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define  boolean  int
#define  bool     int
#define  true     1
```

```
#define  false    0
#define  TRUE     1
#define  FALSE    0
#define  maxint   INT_MAX

#if __MSDOS__ || MSDOS || WIN32 || __WIN32__
#  define  pathsep '\\'
#else
#  define  pathsep '/'
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{ int i;
  char old[256];
  strcpy(old, oldstr);
  i = strlen(old);
  while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
  if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
  if (ext[0] == '.') sprintf(newstr,"%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

#define ASM_alength  8   // maximum length of mnemonics, labels
#define ASM_slength  35  // maximum length of comment and other strings

typedef char ASM_alfa[ASM_alength + 1];
typedef char ASM_strings[ASM_slength + 1];

#include "set.h"

enum ASM_errors {
  ASM_invalidcode, ASM_undefinedlabel, ASM_invalidaddress,
  ASM_unlabelled, ASM_hasaddress, ASM_noaddress,
  ASM_excessfields, ASM_mismatched, ASM_nonalpha,
  ASM_badlabel, ASM_invalidchar, ASM_invalidquote,
  ASM_overflow
};

typedef Set<ASM_overflow> ASM_errorset;

#endif /* MISC_H */


----- set.h ----------------------------------------------------------------

// Simple set operations

#ifndef SET_H
#define SET_H

template <int maxElem>
class Set {                         // { 0 .. maxElem }
  public:
    Set()                           // Construct { }
    { clear(); }

    Set(int e1)                     // Construct { e1 }
    { clear(); incl(e1); }

    Set(int e1, int e2)             // Construct { e1, e2 }
    { clear(); incl(e1); incl(e2); }

    Set(int e1, int e2, int e3)     // Construct { e1, e2, e3 }
    { clear(); incl(e1); incl(e2); incl(e3); }

    Set(int n, int e1[])            // Construct { e[0] .. e[n-1] }
    { clear(); for (int i = 0; i < n; i++) incl(e1[i]); }

    void incl(int e)                // Include e
    { if (e >= 0 && e <= maxElem) bits[wrd(e)] |= bitmask(e); }

    void excl(int e)                // Exclude e
    { if (e >= 0 && e <= maxElem) bits[wrd(e)] &= ~bitmask(e); }

    int memb(int e)                 // Test membership for e
    { if (e >= 0 && e <= maxElem) return((bits[wrd(e)] & bitmask(e)) != 0);
      else return 0;
    }

    int isempty(void)               // Test for empty set
    { for (int i = 0; i < length; i++) if (bits[i]) return 0;
      return 1;
```

```
    }

    Set operator + (const Set &s)   // Union with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] | s.bits[i];
      return r;
    }

    Set operator * (const Set &s)   // Intersection with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] & s.bits[i];
      return r;
    }

    Set operator - (const Set &s)   // Difference with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] & ~s.bits[i];
      return r;
    }

    Set operator / (const Set &s)   // Symmetric difference with s
    { Set<maxElem> r;
      for (int i = 0; i < length; i++) r.bits[i] = bits[i] ^ s.bits[i];
      return r;
    }

  private:
    unsigned char       bits[(maxElem + 8) / 8];
    int                 length;
    int wrd(int i)      { return(i / 8); }
    int bitmask(int i)  { return(1 << (i % 8)); }
    void clear()        { length = (maxElem + 8) / 8;
                          for (int i = 0; i < length; i++) bits[i] = 0;
                        }
};

#endif /* SET_H */


----- sh.s ------------------------------------------------------------------

// Source handler for assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef SH_H
#define SH_H

#include "misc.h"

const int linemax = 129;        // limit on source line length

class SH {
  public:
    FILE *lst; // listing file
    char ch;   // latest character read

    void nextch(void);
    // Returns ch as the next character on current source line, reading a new
    // line where necessary.  ch is returned as NUL if src is exhausted

    bool endline(void)             { return (charpos == linelength); }
    // Returns true when end of current line has been reached

    bool startline(void)           { return (charpos == 1); }
    // Returns true if current ch is the first on a line

    void writehex(int i, int n)    { fprintf(lst, "%02X%*c", i, n-2, ' '); }
    // Writes (byte valued) i to lst file as hex pair, left-justified in n spaces

    void writetext(char *s, int n) { fprintf(lst, "%-*s", n, s); }
    // Writes s to lst file left-justified in n spaces

    SH();
    // Default constructor

    SH(char *sourcename, char *listname, char *version);
    // Initializes source handler, and displays version information on lst file.
    // Opens src and lst files using given names

    ~SH();
    // Closes src and lst files

  private:
```

```
    FILE *src;              // source file
    int charpos;            // character pointer
    int linelength;         // line length
    char line[linemax + 1]; // last line read
};

#endif /*SH_H*/


----- sh.cpp ------------------------------------------------------------------

// Source handler for assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "sh.h"

void SH::nextch(void)
{ if (ch == '\0') return;        // input exhausted
  if (charpos == linelength)     // new line needed
  { linelength = 0; charpos = 0; ch = getc(src);
    while (ch != '\n' && !feof(src))
    { if (linelength < linemax) { line[linelength] = ch; linelength++; }
      ch = getc(src);
    }
    if (feof(src))
      line[linelength] = '\0';   // mark end with an explicit nul
    else
      line[linelength] = ' ';    // mark end with an explicit space
    linelength++;
  }
  ch = line[charpos]; charpos++; // pass back unique character
}

SH::SH(char *sourcename, char *listname, char *version)
{ src = fopen(sourcename, "r");
  if (src == NULL)
    { printf("Could not open input file\n"); exit(1); }
  lst = fopen(listname, "w");
  if (lst == NULL)
    { printf("Could not open listing file\n"); lst = stdout; }
  fprintf(lst, "%s\n\n", version);
  ch = ' '; charpos = 0; linelength = 0;
}

SH::SH()
{ src = NULL; lst = NULL; ch = ' '; charpos = 0; linelength = 0; }

SH::~SH()
{ if (src) fclose(src); src = NULL;
  if (lst) fclose(lst); lst = NULL;
}


----- la.h --------------------------------------------------------------------

// Lexical analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef LA_H
#define LA_H

#include "misc.h"
#include "sh.h"

enum LA_symtypes {
  LA_unknown, LA_eofsym, LA_eolsym, LA_idsym, LA_numsym, LA_comsym,
  LA_commasym, LA_plussym, LA_minussym, LA_starsym
};

struct LA_symbols {
  bool islabel;      // if in first column
  LA_symtypes sym;   // class
  ASM_strings str;   // lexeme
  int num;           // value if numeric
};

class LA {
  public:
    void getsym(LA_symbols &SYM, ASM_errorset &errors);
    // Returns the next symbol on current source line.
    // Adds to set of errors if necessary and returns SYM.sym = unknown
    // if no valid symbol can be recognized
```

```
    LA(SH *S);
    // Associates scanner with source handler S and initializes scanning

  private:
    SH *Srce;
    void getword(LA_symbols &SYM);
    void getnumber(LA_symbols &SYM, ASM_errorset &errors);
    void getcomment(LA_symbols &SYM);
    void getquotedchar(LA_symbols &SYM, char quote, ASM_errorset &errors);
};

#endif /*LA_H*/


----- la.cpp ---------------------------------------------------------------

// Lexical analyzer for assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "la.h"

void LA::getword(LA_symbols &SYM)
// Assemble identifier or opcode, in UPPERCASE for consistency
{ int length = 0;
  while (isalnum(Srce->ch))
  { if (length < ASM_slength)
    { SYM.str[length] = toupper(Srce->ch); length++; }
    Srce->nextch();
  }
  SYM.str[length] = '\0';
}

void LA::getnumber(LA_symbols &SYM, ASM_errorset &errors)
// Assemble number and store its identifier in UPPERCASE for consistency
{ int length = 0;
  while (isdigit(Srce->ch))
  { SYM.num = SYM.num * 10 + Srce->ch - '0';
    if (SYM.num > 255) errors.incl(ASM_overflow);
    SYM.num %= 256;
    if (length < ASM_slength) { SYM.str[length] = toupper(Srce->ch); length++; }
    Srce->nextch();
  }
  SYM.str[length] = '\0';
}

void LA::getcomment(LA_symbols &SYM)
// Assemble comment
{ int length = 0;
  while (!Srce->endline())
  { if (length < ASM_slength) { SYM.str[length] = Srce->ch; length++; }
    Srce->nextch();
  }
  SYM.str[length] = '\0';
}

void LA::getquotedchar(LA_symbols &SYM, char quote, ASM_errorset &errors)
// Assemble single character address token
{ SYM.str[0] = quote;
  Srce->nextch(); SYM.num = Srce->ch; SYM.str[1] = Srce->ch;
  if (!Srce->endline()) Srce->nextch();
  SYM.str[2] = Srce->ch; SYM.str[3] = '\0';
  if (Srce->ch != quote) errors.incl(ASM_invalidquote);
  if (!Srce->endline()) Srce->nextch();
}

void LA::getsym(LA_symbols &SYM, ASM_errorset &errors)
{ SYM.num = 0; SYM.str[0] = '\0';   // empty string
  while (Srce->ch == ' ' && !Srce->endline()) Srce->nextch();
  SYM.islabel = (Srce->startline() && Srce->ch != ' '
                 && Srce->ch != ';' && Srce->ch != '\0');
  if (SYM.islabel && !isalpha(Srce->ch)) errors.incl(ASM_badlabel);
  if (Srce->ch == '\0') { SYM.sym = LA_eofsym; return; }
  if (Srce->endline()) { SYM.sym = LA_eolsym; Srce->nextch(); return; }
  if (isalpha(Srce->ch))
    { SYM.sym = LA_idsym; getword(SYM); }
  else if (isdigit(Srce->ch))
    { SYM.sym = LA_numsym; getnumber(SYM, errors); }
  else switch (Srce->ch)
    { case ';':
        SYM.sym = LA_comsym; getcomment(SYM); break;
      case ',':
        SYM.sym = LA_commasym; strcpy(SYM.str, ","); Srce->nextch(); break;
      case '+':
```

```
        SYM.sym = LA_plussym; strcpy(SYM.str, "+"); Srce->nextch(); break;
      case '-':
        SYM.sym = LA_minussym; strcpy(SYM.str, "-"); Srce->nextch(); break;
      case '*':
        SYM.sym = LA_starsym; strcpy(SYM.str, "*"); Srce->nextch(); break;
      case '\'':
      case '"':
        SYM.sym = LA_numsym; getquotedchar(SYM, Srce->ch, errors); break;
      default:
        SYM.sym = LA_unknown; getcomment(SYM); errors.incl(ASM_invalidchar);
        break;
    }
}

LA::LA(SH* S)
{ Srce = S; Srce->nextch(); }
```

----- sa.h ----------

```
// Syntax analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef SA_H
#define SA_H

#include "misc.h"
#include "la.h"

const int SA_maxterms = 16;

enum SA_termkinds {
  SA_absent, SA_numeric, SA_alphameric, SA_comma, SA_plus, SA_minus, SA_star
};

struct SA_terms {
  SA_termkinds kind;
  int number;      // value if known
  ASM_alfa name;   // character representation
};

struct SA_addresses {
  char length;     // number of fields
  SA_terms term[SA_maxterms - 1];
};

struct SA_unpackedlines {
  // source text, unpacked into fields
  bool labelled;
  ASM_alfa labfield, mnemonic;
  SA_addresses address;
  ASM_strings comment;
  ASM_errorset errors;
};

class SA {
  public:
    void parse(SA_unpackedlines &srcline);
    // Analyzes the next source line into constituent fields

    SA(LA *L);
    // Associates syntax analyzer with its lexical analyzer L

  private:
    LA *Lex;
    LA_symbols SYM;
    void GetSym(ASM_errorset &errors);
    void getaddress(SA_unpackedlines &srcline);
};

#endif /*SA_H*/
```

----- sa.cpp ----------

```
// Syntax analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "sa.h"
#include "set.h"

typedef Set<LA_starsym> symset;
```

```
void SA::GetSym(ASM_errorset &errors)
{ Lex->getsym(SYM, errors); }

void SA::getaddress(SA_unpackedlines &srcline)
// Unpack the addressfield of line into srcline
{ symset allowed(LA_idsym, LA_numsym, LA_starsym);
  symset possible = allowed + symset(LA_commasym, LA_plussym, LA_minussym);
  srcline.address.length = 0;
  while (possible.memb(SYM.sym))
  { if (!allowed.memb(SYM.sym))
      srcline.errors.incl(ASM_invalidaddress);
    if (srcline.address.length < SA_maxterms - 1)
      srcline.address.length++;
    else
      srcline.errors.incl(ASM_excessfields);
    sprintf(srcline.address.term[srcline.address.length - 1].name, "%.*s",
            ASM_alength, SYM.str);
    srcline.address.term[srcline.address.length - 1].number = SYM.num;
    switch (SYM.sym)
    { case LA_numsym:
        srcline.address.term[srcline.address.length - 1].kind = SA_numeric;
        break;
      case LA_idsym:
        srcline.address.term[srcline.address.length - 1].kind = SA_alphameric;
        break;
      case LA_plussym:
        srcline.address.term[srcline.address.length - 1].kind = SA_plus;
        break;
      case LA_minussym:
        srcline.address.term[srcline.address.length - 1].kind = SA_minus;
        break;
      case LA_starsym:
        srcline.address.term[srcline.address.length - 1].kind = SA_star;
        break;
      case LA_commasym:
        srcline.address.term[srcline.address.length - 1].kind = SA_comma;
        break;
    }
    allowed = possible - allowed;
    GetSym(srcline.errors); // check trailing comment, parameters
  }
  if (!(srcline.address.length & 1)) srcline.errors.incl(ASM_invalidaddress);
}

void SA::parse(SA_unpackedlines &srcline)
{ symset startaddress(LA_idsym, LA_numsym, LA_starsym);
  srcline.labfield[0] = '\0';
  strcpy(srcline.mnemonic, "   ");
  srcline.comment[0] = '\0';
  srcline.errors = ASM_errorset();
  srcline.address.term[0].kind = SA_absent;
  srcline.address.term[0].number = 0;
  srcline.address.term[0].name[0] = '\0';
  srcline.address.length = 0;
  GetSym(srcline.errors);       // first on line - opcode or label ?
  if (SYM.sym == LA_eofsym) { strcpy(srcline.mnemonic, "END"); return; }
  srcline.labelled = SYM.islabel;
  if (srcline.labelled)         // must look for the opcode
  { srcline.labelled = srcline.errors.isempty();
    sprintf(srcline.labfield, "%.*s", ASM_alength, SYM.str);
    GetSym(srcline.errors);     // probably an opcode
  }
  if (SYM.sym == LA_idsym)      // has a mnemonic
  { sprintf(srcline.mnemonic, "%.*s", ASM_alength, SYM.str);
    GetSym(srcline.errors);     // possibly an address
    if (startaddress.memb(SYM.sym)) getaddress(srcline);
  }
  if (SYM.sym == LA_comsym || SYM.sym == LA_unknown)
  { strcpy(srcline.comment, SYM.str); GetSym(srcline.errors); }
  if (SYM.sym != LA_eolsym)     // spurious symbol
  { strcpy(srcline.comment, SYM.str); srcline.errors.incl(ASM_excessfields); }
  while (SYM.sym != LA_eolsym && SYM.sym != LA_eofsym)
    GetSym(srcline.errors);     // consume garbage
}

SA::SA(LA * L)
{ Lex = L; }


----- st.h ----------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using simple linked list
```

```
// P.D. Terry, Rhodes University, 1996

#ifndef ST_H
#define ST_H

#include "misc.h"
#include "mc.h"
#include "sh.h"

enum ST_actions { ST_add, ST_subtract };

typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v, ST_actions a);

struct ST_forwardrefs {   // forward references for undefined labels
  MC_bytes byte;          // to be patched
  ST_actions action;      // taken when patching
  ST_forwardrefs *nlink;  // to next reference
};

struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool defined;           // true after defining occurrence encountered
  ST_entries *slink;      // to next entry
  ST_forwardrefs *flink;  // to forward references
};

class ST {
  public:
    void printsymboltable(bool &errors);
    // Summarizes symbol table at end of assembly, and alters errors to true if
    // any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // Records action to be applied later in fixing up forward references.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes *mem, ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)

  private:
    SH *Srce;
    ST_entries *lastsym;
    void findentry(ST_entries *&symentry, char *name, bool &found);
};

#endif /*ST_H*/


----- st.cpp ----------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using simply linked list
// P.D. Terry, Rhodes University, 1996

#include "st.h"

void ST::printsymboltable(bool &errors)
{ fprintf(Srce->lst, "\nSymbol Table\n");
  fprintf(Srce->lst, "------------\n");
  ST_entries *symentry = lastsym;
  while (symentry)
  { Srce->writetext(symentry->name, 10);
    if (!symentry->defined)
    { fprintf(Srce->lst, " --- undefined"); errors = true; }
    else
    { Srce->writehex(symentry->value, 3);
      fprintf(Srce->lst, "%5d", symentry->value);
    }
    putc('\n', Srce->lst);
    symentry = symentry->slink;
  }
  putc('\n', Srce->lst);
}
```

```
void ST::findentry(ST_entries *&symentry, char *name, bool &found)
{ symentry = lastsym;
  found = false;
  while (!found && symentry)
  { if (!strcmp(name, symentry->name))
      found = true;
    else
      symentry = symentry->slink;
  }
  if (found) return;
  symentry = new ST_entries;  // make new forward reference entry
  sprintf(symentry->name, "%.*s", ASM_alength, name);
  symentry->value = 0;
  symentry->defined = false;
  symentry->flink = NULL;
  symentry->slink = lastsym;
  lastsym = symentry;
}

void ST::enter(char *name, MC_bytes value)
{ ST_entries *symentry;
  bool found;
  findentry(symentry, name, found);
  symentry->value = value;
  symentry->defined = true;
}

void ST::valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined)
{ ST_entries *symentry;
  ST_forwardrefs *forwardentry;
  bool found;
  findentry(symentry, name, found);
  value = symentry->value;
  undefined = !symentry->defined;
  if (!undefined) return;
  forwardentry = new ST_forwardrefs; // new node in reference chain
  forwardentry->byte = location; forwardentry->action = action;
  if (found)                         // it was already in the table
    forwardentry->nlink = symentry->flink;
  else                               // new entry in the table
    forwardentry->nlink = NULL;
  symentry->flink = forwardentry;
}

void ST::outstandingreferences(MC_bytes mem[], ST_patch fix)
{ ST_forwardrefs *link;
  ST_entries *symentry = lastsym;
  while (symentry)
  { link = symentry->flink;
    while (link)
    { fix(mem, link->byte, symentry->value, link->action);
      link = link->nlink;
    }
    symentry = symentry->slink;
  }
}

ST::ST(SH *S)
{ Srce = S; lastsym = NULL; }


----- st.h ----------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using hashing technique with collision stepping
// P.D. Terry, Rhodes University, 1996

#ifndef ST_H
#define ST_H

#include "misc.h"
#include "mc.h"
#include "sh.h"

const int tablemax = 239;   // symbol table size
const int tablestep = 7;    // a prime number

enum ST_actions { ST_add, ST_subtract };

typedef void (*ST_patch)(MC_bytes mem[], MC_bytes b, MC_bytes v, ST_actions a);
typedef short tableindex;
```

```
struct ST_forwardrefs {  // forward references for undefined labels
  MC_bytes byte;         // to be patched
  ST_actions action;     // taken when patching
  ST_forwardrefs *nlink; // to next reference
};

struct ST_entries {
  ASM_alfa name;          // name
  MC_bytes value;         // value once defined
  bool used;              // true when in use already
  bool defined;           // true after defining occurrence encountered
  ST_forwardrefs *flink;  // to forward references
};

class ST {
  public:
    void printsymboltable(bool &errors);
    // Summarizes symbol table at end of assembly, and alters errors
    // to true if any symbols have remained undefined

    void enter(char *name, MC_bytes value);
    // Adds name to table with known value

    void valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined);
    // Returns value of required name, and sets undefined if not found.
    // Records action to be applied later in fixing up forward references.
    // location is the current value of the instruction location counter

    void outstandingreferences(MC_bytes mem[], ST_patch fix);
    // Walks symbol table, applying fix to outstanding references in mem

    ST(SH *S);
    // Associates table handler with source handler S (for listings)

  private:
    SH *Srce;
    ST_entries hashtable[tablemax + 1];
    void findentry(tableindex &symentry, char *name, bool &found);
};

#endif /*ST_H*/


----- st.cpp ----------------------------------------------------------------

// Table handler for one-pass macro assembler for single-accumulator machine
// Version using hashing technique with collision stepping
// P.D. Terry, Rhodes University, 1996

#include "st.h"

void ST::printsymboltable(bool &errors)
{ fprintf(Srce->lst, "\nSymbol Table\n");
  fprintf(Srce->lst, "------------\n");
  for (tableindex i = 0; i < tablemax; i++)
  { if (hashtable[i].used)
    { Srce->writetext(hashtable[i].name, 10);
      if (!hashtable[i].defined)
      { fprintf(Srce->lst, " --- undefined"); errors = true; }
      else
      { Srce->writehex(hashtable[i].value, 3);
        fprintf(Srce->lst, "%5d", hashtable[i].value);
      }
      putc('\n', Srce->lst);
    }
  }
  putc('\n', Srce->lst);
}

tableindex hashkey(char *ident)
{ const int large = (maxint - 256); // large number in hashing function
  int sum = 0, l = strlen(ident);
  for (int i = 0; i < l; i++) sum = (sum + ident[i]) % large;
  return (sum % tablemax);
}

void ST::findentry(tableindex &symentry, char *name, bool &found)
{ enum { looking, entered, caninsert, overflow } state;
  symentry = hashkey(name);
  state = looking;
  tableindex start = symentry;
  while (state == looking)
```

```
  { if (!hashtable[symentry].used)
      { state = caninsert; break; }
    if (!strcmp(name, hashtable[symentry].name))
      { state = entered; break; }
    symentry = (symentry + tablestep) % tablemax;
    if (symentry == start) state = overflow;
  }
  switch (state)
  { case caninsert:
      sprintf(hashtable[symentry].name, "%.*s", ASM_alength, name);
      hashtable[symentry].value = 0;
      hashtable[symentry].used = true;
      hashtable[symentry].flink = NULL;
      hashtable[symentry].defined = false;
      break;
    case overflow:
      printf("Symbol table overflow\n");
      exit(1);
      break;
    case entered:   // no further action
      break;
  }
  found = (state == entered);
}

void ST::enter(char *name, MC_bytes value)
{ tableindex symentry;
  bool found;
  findentry(symentry, name, found);
  hashtable[symentry].value = value;
  hashtable[symentry].defined = true;
}

void ST::valueofsymbol(char *name, MC_bytes location, MC_bytes &value,
                       ST_actions action, bool &undefined)
{ tableindex symentry;
  ST_forwardrefs *forwardentry;
  bool found;
  findentry(symentry, name, found);
  value = hashtable[symentry].value;
  undefined = !hashtable[symentry].defined;
  if (!undefined) return;
  forwardentry = new ST_forwardrefs; // new node in reference chain
  forwardentry->byte = location;
  forwardentry->action = action;
  if (found)                         // it was already in the table
    forwardentry->nlink = hashtable[symentry].flink;
  else                               // new entry in the table
    forwardentry->nlink = NULL;
  hashtable[symentry].flink = forwardentry;
}

void ST::outstandingreferences(MC_bytes mem[], ST_patch fix)
{ ST_forwardrefs *link;
  for (tableindex i = 0; i < tablemax; i++)
  { if (hashtable[i].used)
    { link = hashtable[i].flink;
      while (link)
      { fix(mem, link->byte, hashtable[i].value, link->action);
        link = link->nlink;
      }
    }
  }
}

ST::ST(SH *S)
{ Srce = S;
  for (tableindex i = 0; i < tablemax; i++) hashtable[i].used = false;
}


----- mh.h ------------------------------------------------------------------

// Macro analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef MH_H
#define MH_H

#include "asmbase.h"

typedef struct MH_macentries *MH_macro;
```

```
class MH {
  public:
    void newmacro(MH_macro &m, SA_unpackedlines header);
    // Creates m as a new macro, with given header line that includes the
    // formal parameters

    void storeline(MH_macro m, SA_unpackedlines line);
    // Adds line to the definition of macro m

    void checkmacro(char *name, MH_macro &m, bool &ismacro, int &params);
    // Checks to see whether name is that of a predefined macro.  Returns
    // ismacro as the result of the search.  If successful, returns m as
    // the macro, and params as the number of formal parameters

    void expand(MH_macro m, SA_addresses actualparams,
                ASMBASE *assembler, bool &errors);
    // Expands macro m by invoking assembler for each line of the macro
    // definition, and using the actualparams supplied in place of the
    // formal parameters appearing in the macro header.
    // errors is altered to true if the assembly fails for any reason

    MH();
    // Initializes macro handler

  private:
    MH_macro lastmac;
    int position(MH_macro m, char *str);
    void substituteactualparameters(MH_macro m,
                                    SA_addresses actualparams,
                                    SA_unpackedlines &nextline);
};

#endif /*MH_H*/


----- mh.cpp ----------------------------------------------------------------

// Macro analyzer for macro assemblers for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "mh.h"

struct MH_lines {
  SA_unpackedlines text;          // a single line of macro text
  MH_lines *link;                 // link to the next line in the macro
};

struct MH_macentries {
  SA_unpackedlines definition;    // header line
  MH_macro mlink;                 // link to next macro in list
  MH_lines *firstline, *lastline; // links to the text of this macro
};

void MH::newmacro(MH_macro &m, SA_unpackedlines header)
{ m = new MH_macentries;
  m->definition = header;         // store formal parameters
  m->firstline = NULL;            // no text yet
  m->mlink = lastmac;             // link to rest of macro definitions
  lastmac = m;                    // and this becomes the last macro added
}

void MH::storeline(MH_macro m, SA_unpackedlines line)
{ MH_lines *newline = new MH_lines;
  newline->text = line;           // store source line
  newline->link = NULL;           // at the end of the queue
  if (m->firstline == NULL)       // first line of macro?
    m->firstline = newline;       // form head of new queue
  else
    m->lastline->link = newline;  // add to tail of existing queue
  m->lastline = newline;
}

void MH::checkmacro(char *name, MH_macro &m, bool &ismacro, int &params)
{ m = lastmac; ismacro = false; params = 0;
  while (m && !ismacro)
  { if (!strcmp(name, m->definition.labfield))
      { ismacro = true; params = m->definition.address.length; }
    else
      m = m->mlink;
  }
}
```

```
int MH::position(MH_macro m, char *str)
// Search formals for match to str; returns 0 if no match
{ bool found = false;
  int i = m->definition.address.length - 1;
  while (i >= 0 && !found)
  { if (!strcmp(str, m->definition.address.term[i].name))
      found = true;
    else
      i--;
  }
  return i;
}

void MH::substituteactualparameters(MH_macro m,
         SA_addresses actualparams, SA_unpackedlines &nextline)
// Substitute label, mnemonic or address components into
// nextline where necessary
{ int j = 0, i = position(m, nextline.labfield); // check label
  if (i >= 0) strcpy(nextline.labfield, actualparams.term[i].name);
  i = position(m, nextline.mnemonic);            // check mnemonic
  if (i >= 0) strcpy(nextline.mnemonic, actualparams.term[i].name);
  j = 0;                                         // check address fields
  while (j < nextline.address.length)
  { i = position(m, nextline.address.term[j].name);
    if (i >= 0) nextline.address.term[j] = actualparams.term[i];
    j += 2;                                      // bypass commas
  }
}

void MH::expand(MH_macro m, SA_addresses actualparams,
                ASMBASE *assembler, bool &errors)
{ SA_unpackedlines nextline;
  if (!m) return;                                // nothing to do
  MH_lines *current = m->firstline;
  while (current)
  { nextline = current->text;                    // retrieve line of macro text
    substituteactualparameters(m, actualparams, nextline);
    assembler->assembleline(nextline, errors);   // and asssemble it
    current = current->link;
  }
}

MH::MH()
{ lastmac = NULL; }


----- asmbase.h ----------------------------------------------------------------

// Base assembler class for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef ASMBASE_H
#define ASMBASE_H

#include "misc.h"
#include "sa.h"

class ASMBASE {
  public:
    virtual void assembleline(SA_unpackedlines &srcline, bool &failure) = 0;
    // Assemble srcline, reporting failure if it occurs
};

#endif /*A_H*/


----- as.h ---------------------------------------------------------------------

// One-pass macro assembler for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#ifndef AS_H
#define AS_H

#include "asmbase.h"
#include "mc.h"
#include "st.h"
#include "sh.h"
#include "mh.h"

class AS : ASMBASE {
  public:
    void assemble(bool &errors);
```

```
    // Assembles and lists program.
    // Assembled code is dumped to file for later interpretation, and left
    // in pseudo-machine memory for immediate interpretation if desired.
    // Returns errors = true if assembly fails

    virtual void assembleline(SA_unpackedlines &srcline, bool &failure);
    // Assemble srcline, reporting failure if it occurs

    AS(char *sourcename, char *listname, char *version, MC *M);
    // Instantiates version of the assembler to process sourcename, creating
    // listings in listname, and generating code for associated machine M

  private:
    SH *Srce;
    LA *Lex;
    SA *Parser;
    ST *Table;
    MC *Machine;
    MH *Macro;

    struct { ASM_alfa spelling; MC_bytes byte; } optable[256];
    int opcodes;         // number of opcodes actually defined
    struct objlines { MC_bytes location, opcode, address; };
    objlines objline;    // current line as assembled
    MC_bytes location;   // location counter
    bool assembling;     // monitor progress of assembly
    bool include;        // handle conditional assembly

    MC_bytes bytevalue(char *mnemonic);
    void enter(char *mnemonic, MC_bytes thiscode);
    void termvalue(SA_terms term, MC_bytes &value, ST_actions action,
                   bool &undefined, bool &badaddress);
    void evaluate(SA_addresses address, MC_bytes &value,
                  bool &undefined, bool &malformed);

    void listerrors(ASM_errorset allerrors, bool &failure);
    void listcode(void);
    void listsourceline(SA_unpackedlines &srcline, bool coderequired,
                        bool &failure);
    void definemacro(SA_unpackedlines &srcline, bool &failure);
    void firstpass(bool &errors);
};

#endif /*AS_H*/


----- as.cpp ------------------------------------------------------------------

// One-pass macro assembler for the single-accumulator machine
// P.D. Terry, Rhodes University, 1996

#include "as.h"

const bool nocodelisted = false;
const bool codelisted = true;

enum directives {
  AS_err = 61,  // erroneous opcode
  AS_nul = 62,  // blank opcode
  AS_beg = 63,  // introduce program
  AS_end = 64,  // end of source
  AS_mac = 65,  // introduce macro
  AS_ds  = 66,  // define storage
  AS_equ = 67,  // equate
  AS_org = 68,  // set location counter
  AS_if  = 69,  // conditional
  AS_dc  = 70   // define constant byte
};

MC_bytes AS::bytevalue(char *mnemonic)
{ int look, l = 1, r = opcodes;
  do                                // binary search
  { look = (l + r) / 2;
    if (strcmp(mnemonic, optable[look].spelling) <= 0) r = look - 1;
    if (strcmp(mnemonic, optable[look].spelling) >= 0) l = look + 1;
  } while (l <= r);
  if (l > r + 1)
    return (optable[look].byte);  // found it
  else
    return (optable[0].byte);     // err entry
}

void AS::enter(char *mnemonic, MC_bytes thiscode)
```

```
// Add (mnemonic, thiscode) to optable for future look up
{ strcpy(optable[opcodes].spelling, mnemonic);
  optable[opcodes].byte = thiscode;
  opcodes++;
}

void backpatch(MC_bytes mem[], MC_bytes location, MC_bytes value, ST_actions how)
{ switch (how)
  { case ST_add:
      mem[location] = (mem[location] + value) % 256; break;
    case ST_subtract:
      mem[location] = (mem[location] - value + 256) % 256; break;
  }
}

void AS::termvalue(SA_terms term, MC_bytes &value, ST_actions action,
                   bool &undefined, bool &badaddress)
// Determine value of a single term, recording outstanding action
// if undefined so far, and recording badaddress if malformed
{ undefined = false;
  switch (term.kind)
  { case SA_absent:
    case SA_numeric:
      value = term.number % 256; break;
    case SA_star:
      value = location; break;
    case SA_alphameric:
      Table->valueofsymbol(term.name, location, value, action, undefined); break;
    default:
      badaddress = true; value = 0; break;
  }
}

void AS::evaluate(SA_addresses address, MC_bytes &value, bool &undefined,
                  bool &malformed)
// Determine value of address, recording whether undefined or malformed
{ ST_actions nextaction;
  MC_bytes nextvalue;
  bool unknown;
  malformed = false;
  termvalue(address.term[0], value, ST_add, undefined, malformed);
  int i = 1;
  while (i < address.length)
  { switch (address.term[i].kind)
    { case SA_plus:  nextaction = ST_add; break;
      case SA_minus: nextaction = ST_subtract; break;
      default:       nextaction = ST_add; malformed = true; break;
    }
    i++;
    termvalue(address.term[i], nextvalue, nextaction, unknown, malformed);
    switch (nextaction)
    { case ST_add:      value = (value + nextvalue) % 256; break;
      case ST_subtract: value = (value - nextvalue + 256) % 256; break;
    }
    undefined = (undefined || unknown);
    i++;
  }
}

static char *ErrorMsg[] = {
  " - unknown opcode",
  " - address field not resolved",
  " - invalid address field",
  " - label missing",
  " - spurious address field",
  " - address field missing",
  " - address field too long",
  " - wrong number of parameters",
  " - invalid formal parameters",
  " - invalid label",
  " - unknown character",
  " - mismatched quotes",
  " - number too large",
};

void AS::listerrors(ASM_errorset allerrors, bool &failure)
{ if (allerrors.isempty()) return;
  failure = true;
  fprintf(Srce->lst, "Next line has errors");
  for (int error = ASM_invalidcode; error <= ASM_overflow; error++)
    if (allerrors.memb(error)) fprintf(Srce->lst, "%s\n", ErrorMsg[error]);
}
```

```
void AS::listcode(void)
// List generated code bytes on source listing
{ Srce->writehex(objline.location, 4);
  if (objline.opcode >= AS_err && objline.opcode <= AS_if)
    fprintf(Srce->lst, "       ");
  else if (objline.opcode <= MC_hlt)              // OneByteOps
    Srce->writehex(objline.opcode, 7);
  else if (objline.opcode == AS_dc)               // DC special case
    Srce->writehex(objline.address, 7);
  else                                            // TwoByteOps
  { Srce->writehex(objline.opcode, 3);
    Srce->writehex(objline.address, 4);
  }
}

void AS::listsourceline(SA_unpackedlines &srcline, bool coderequired,
                        bool &failure)
// List srcline, with option of listing generated code
{ listerrors(srcline.errors, failure);
  if (coderequired) listcode(); else fprintf(Srce->lst, "           ");
  Srce->writetext(srcline.labfield, 9);
  Srce->writetext(srcline.mnemonic, 9);
  int width = strlen(srcline.address.term[0].name);
  fputs(srcline.address.term[0].name, Srce->lst);
  for (int i = 1; i < srcline.address.length; i++)
  { width += strlen(srcline.address.term[i].name) + 1;
    putc(' ', Srce->lst);
    fputs(srcline.address.term[i].name, Srce->lst);
  }
  if (width < 30) Srce->writetext(" ", 30 - width);
  fprintf(Srce->lst, "%s\n", srcline.comment);
}

void AS::definemacro(SA_unpackedlines &srcline, bool &failure)
// Handle introduction of a macro (possibly nested)
{ MC_bytes opcode;
  MH_macro macro;
  bool declared = false;
  int i = 0;
  if (srcline.labelled)                           // name must be present
    declared = true;
  else
    srcline.errors.incl(ASM_unlabelled);
  if (!(srcline.address.length & 1))              // must be an odd number of terms
    srcline.errors.incl(ASM_invalidaddress);
  while (i < srcline.address.length)              // check that formals are names
  { if (srcline.address.term[i].kind != SA_alphameric)
      srcline.errors.incl(ASM_nonalpha);
    i += 2;                                       // bypass commas
  }
  listsourceline(srcline, nocodelisted, failure);
  if (declared) Macro->newmacro(macro, srcline);  // store header
  do
  { Parser->parse(srcline);                       // next line of macro text
    opcode = bytevalue(srcline.mnemonic);
    if (opcode == AS_mac)                         // nested macro?
      definemacro(srcline, failure);              // recursion handles it
    else
    { listsourceline(srcline, nocodelisted, failure);
      if (declared && opcode != AS_end && srcline.errors.isempty())
        Macro->storeline(macro, srcline);         // add to macro text
    }
  } while (opcode != AS_end);
}

void AS::assembleline(SA_unpackedlines &srcline, bool &failure)
// Assemble single srcline
{ if (!include) { include = true; return; }       // conditional assembly
  bool badaddress, found, undefined;
  MH_macro macro;
  int formal;
  Macro->checkmacro(srcline.mnemonic, macro, found, formal);
  if (found)                                      // expand macro and exit
  { if (srcline.labelled) Table->enter(srcline.labfield, location);
    if (formal != srcline.address.length)         // number of params okay?
      srcline.errors.incl(ASM_mismatched);
    listsourceline(srcline, nocodelisted, failure);
    if (srcline.errors.isempty())                 // okay to expand?
      Macro->expand(macro, srcline.address, this, failure);
    return;
  }
  badaddress = false;
  objline.location = location; objline.address = 0;
```

```
  objline.opcode = bytevalue(srcline.mnemonic);
  if (objline.opcode == AS_err)                  // check various constraints
    srcline.errors.incl(ASM_invalidcode);
  else if (objline.opcode > AS_mac ||
           objline.opcode > MC_hlt && objline.opcode < AS_err)
    { if (srcline.address.length == 0) srcline.errors.incl(ASM_noaddress); }
  else if (objline.opcode != AS_mac && srcline.address.length != 0)
    srcline.errors.incl(ASM_hasaddress);
  if (objline.opcode >= AS_err && objline.opcode <= AS_dc)
  { switch (objline.opcode)                      // directives
    { case AS_beg:
        location = 0;
        break;
      case AS_org:
        evaluate(srcline.address, location, undefined, badaddress);
        if (undefined) srcline.errors.incl(ASM_undefinedlabel);
        objline.location = location;
        break;
      case AS_ds:
        if (srcline.labelled) Table->enter(srcline.labfield, location);
        evaluate(srcline.address, objline.address, undefined, badaddress);
        if (undefined) srcline.errors.incl(ASM_undefinedlabel);
        location = (location + objline.address) % 256;
        break;
      case AS_nul:
      case AS_err:
        if (srcline.labelled) Table->enter(srcline.labfield, location);
        break;
      case AS_equ:
        evaluate(srcline.address, objline.address, undefined, badaddress);
        if (srcline.labelled)
          Table->enter(srcline.labfield, objline.address);
        else
          srcline.errors.incl(ASM_unlabelled);
        if (undefined) srcline.errors.incl(ASM_undefinedlabel);
        break;
      case AS_dc:
        if (srcline.labelled) Table->enter(srcline.labfield, location);
        evaluate(srcline.address, objline.address, undefined, badaddress);
        Machine->mem[location] = objline.address;
        location = (location + 1) % 256;
        break;
      case AS_if:
        evaluate(srcline.address, objline.address, undefined, badaddress);
        if (undefined) srcline.errors.incl(ASM_undefinedlabel);
        include = (objline.address != 0);
        break;
      case AS_mac:
        definemacro(srcline, failure);
        break;
      case AS_end:
        assembling = false;
        break;
    }
  }
  else                                           // machine ops
  { if (srcline.labelled) Table->enter(srcline.labfield, location);
    Machine->mem[location] = objline.opcode;
    if (objline.opcode > MC_hlt)                 // TwoByteOps
    { location = (location + 1) % 256;
      evaluate(srcline.address, objline.address, undefined, badaddress);
      Machine->mem[location] = objline.address;
    }
    location = (location + 1) % 256;             // bump location counter
  }
  if (badaddress) srcline.errors.incl(ASM_invalidaddress);
  if (objline.opcode != AS_mac) listsourceline(srcline, codelisted, failure);
}

void AS::firstpass(bool &errors)
// Make first and only pass over source code
{ SA_unpackedlines srcline;
  location = 0; assembling = true; include = true; errors = false;
  while (assembling)
    { Parser->parse(srcline); assembleline(srcline, errors); }
  Table->printsymboltable(errors);
  if (!errors) Table->outstandingreferences(Machine->mem, backpatch);
}

void AS::assemble(bool &errors)
{ printf("Assembling ...\n");
  fprintf(Srce->lst, "(One Pass Macro Assembler)\n\n");
  firstpass(errors);
```

```
  Machine->listcode();
}

AS::AS(char *sourcename, char *listname, char *version, MC *M)
{ Machine = M;
  Srce    = new SH(sourcename, listname, version);
  Lex     = new LA(Srce);
  Parser  = new SA(Lex);
  Table   = new ST(Srce);
  Macro   = new MH();
  // enter opcodes and mnemonics in ALPHABETIC order
  // done this way for ease of modification later
  opcodes = 0;   // bogus one for erroneous data
  enter("Error   ", AS_err);   // for lines with no opcode
  enter("   ", AS_nul); enter("ACI", MC_aci); enter("ACX", MC_acx);
  enter("ADC", MC_adc); enter("ADD", MC_add); enter("ADI", MC_adi);
  enter("ADX", MC_adx); enter("ANA", MC_ana); enter("ANI", MC_ani);
  enter("ANX", MC_anx); enter("BCC", MC_bcc); enter("BCS", MC_bcs);
  enter("BEG", AS_beg); enter("BNG", MC_bng); enter("BNZ", MC_bnz);
  enter("BPZ", MC_bpz); enter("BRN", MC_brn); enter("BZE", MC_bze);
  enter("CLA", MC_cla); enter("CLC", MC_clc); enter("CLX", MC_clx);
  enter("CMC", MC_cmc); enter("CMP", MC_cmp); enter("CPI", MC_cpi);
  enter("CPX", MC_cpx); enter("DC",  AS_dc);  enter("DEC", MC_dec);
  enter("DEX", MC_dex); enter("DS",  AS_ds);  enter("END", AS_end);
  enter("EQU", AS_equ); enter("HLT", MC_hlt); enter("IF",  AS_if);
  enter("INA", MC_ina); enter("INB", MC_inb); enter("INC", MC_inc);
  enter("INH", MC_inh); enter("INI", MC_ini); enter("INX", MC_inx);
  enter("JSR", MC_jsr); enter("LDA", MC_lda); enter("LDI", MC_ldi);
  enter("LDX", MC_ldx); enter("LSI", MC_lsi); enter("LSP", MC_lsp);
  enter("MAC", AS_mac); enter("NOP", MC_nop); enter("ORA", MC_ora);
  enter("ORG", AS_org); enter("ORI", MC_ori); enter("ORX", MC_orx);
  enter("OTA", MC_ota); enter("OTB", MC_otb); enter("OTC", MC_otc);
  enter("OTH", MC_oth); enter("OTI", MC_oti); enter("POP", MC_pop);
  enter("PSH", MC_psh); enter("RET", MC_ret); enter("SBC", MC_sbc);
  enter("SBI", MC_sbi); enter("SBX", MC_sbx); enter("SCI", MC_sci);
  enter("SCX", MC_scx); enter("SHL", MC_shl); enter("SHR", MC_shr);
  enter("STA", MC_sta); enter("STX", MC_stx); enter("SUB", MC_sub);
  enter("TAX", MC_tax);
}


----- mc.h ----------------------------------------------------------------

// Definition of simple single-accumulator machine and simple emulator
// P.D. Terry, Rhodes University, 1996

#ifndef MC_H
#define MC_H

#include "misc.h"

// machine instructions - order important
enum MC_opcodes {
  MC_nop, MC_cla, MC_clc, MC_clx, MC_cmc, MC_inc, MC_dec, MC_inx, MC_dex,
  MC_tax, MC_ini, MC_inh, MC_inb, MC_ina, MC_oti, MC_otc, MC_oth, MC_otb,
  MC_ota, MC_psh, MC_pop, MC_shl, MC_shr, MC_ret, MC_hlt, MC_lda, MC_ldx,
  MC_ldi, MC_lsp, MC_lsi, MC_sta, MC_stx, MC_add, MC_adx, MC_adi, MC_adc,
  MC_acx, MC_aci, MC_sub, MC_sbx, MC_sbi, MC_sbc, MC_scx, MC_sci, MC_cmp,
  MC_cpx, MC_cpi, MC_ana, MC_anx, MC_ani, MC_ora, MC_orx, MC_ori, MC_brn,
  MC_bze, MC_bnz, MC_bpz, MC_bng, MC_bcc, MC_bcs, MC_jsr, MC_bad = 255 };

typedef enum { running, finished, nodata, baddata, badop } status;
typedef unsigned char MC_bytes;

class MC {
  public:
    MC_bytes mem[256];    // virtual machine memory

    void listcode(void);
    // Lists the 256 bytes stored in mem on requested output file

    void emulator(MC_bytes initpc, FILE *data, FILE *results, bool tracing);
    // Emulates action of the instructions stored in mem, with program counter
    // initialized to initpc.  data and results are used for I/O.
    // Tracing at the code level may be requested

    void interpret(void);
    // Interactively opens data and results files, and requests entry point.
    // Then interprets instructions stored in MC_mem

    MC_bytes opcode(char *str);
    // Maps str to opcode, or to MC_bad (0FFH) if no match can be found
```

```
    MC();
    // Initializes accumulator machine

  private:
    struct processor {
      MC_bytes a;      // Accumulator
      MC_bytes sp;     // Stack pointer
      MC_bytes x;      // Index register
      MC_bytes ir;     // Instruction register
      MC_bytes pc;     // Program count
      bool z, p, c;    // Condition flags
    };
    processor cpu;
    status ps;

    char *mnemonics[256];
    void trace(FILE *results, MC_bytes pcnow);
    void postmortem(FILE *results, MC_bytes pcnow);
    void setflags(MC_bytes MC_register);
    MC_bytes index(void);
};

#endif /*MC_H*/


----- mc.cpp ------------------------------------------------------------------

// Definition of simple single-accumulator machine and simple emulator
// P.D. Terry, Rhodes University, 1996

#include "misc.h"
#include "mc.h"

// set break-in character as CTRL-A (cannot easily use \033 on MS-DOS)
const int ESC = 1;

inline void increment(MC_bytes &x)
// Increment with folding at 256
{ x = (x + 257) % 256; }

inline void decrement(MC_bytes &x)
// Decrement with folding at 256
{ x = (x + 255) % 256; }

MC_bytes MC::opcode(char *str)
// Simple linear search suffices for illustration
{ for (int i = 0; str[i]; i++) str[i] = toupper(str[i]);
  MC_bytes l = MC_nop;
  while (l <= MC_jsr && strcmp(str, mnemonics[l])) l++;
  if (l <= MC_jsr) return l; else return MC_bad;
}

void MC::listcode(void)
// Simply print all 256 bytes in 16 rows
{ MC_bytes nextbyte = 0;
  char filename[256];
  printf("Listing code ... \n");
  printf("Listing file [NUL] ? ");
  gets(filename);
  if (*filename == '\0') return;
  FILE *listfile = fopen(filename, "w");
  if (listfile == NULL) listfile = stdout;
  putc('\n', listfile);
  for (int i = 1; i <= 16; i++)
  { for (int j = 1; j <= 16; j++)
    { fprintf(listfile, "%4d", mem[nextbyte]); increment(nextbyte); }
    putc('\n', listfile);
  }
  if (listfile != stdout) fclose(listfile);
}

void MC::trace(FILE *results, MC_bytes pcnow)
// Simple trace facility for run time debugging
{ fprintf(results, " PC = %02X  A = %02X  ", pcnow, cpu.a);
  fprintf(results, " X = %02X  SP = %02X  ", cpu.x, cpu.sp);
  fprintf(results, " Z = %d P = %d C = %d", cpu.z, cpu.p, cpu.c);
  fprintf(results, " OPCODE = %02X  (%s)\n", cpu.ir, mnemonics[cpu.ir]);
}

void MC::postmortem(FILE *results, MC_bytes pcnow)
// Report run time error and position
{ switch (ps)
  { case badop:   fprintf(results, "Illegal opcode"); break;
```

```
    case nodata:  fprintf(results, "No more data"); break;
    case baddata: fprintf(results, "Invalid data"); break;
  }
  fprintf(results, " at %d\n", pcnow);
  trace(results, pcnow);
  printf("\nPress RETURN to continue\n");
  scanf("%*[^\n]"); getchar();
  listcode();
}

inline void MC::setflags(MC_bytes MC_register)
// Set P and Z flags according to contents of register
{ cpu.z = (MC_register == 0); cpu.p = (MC_register <= 127); }

inline MC_bytes MC::index(void)
// Get indexed address with folding at 256
{ return ((mem[cpu.pc] + cpu.x) % 256); }

void readchar(FILE *data, char &ch, status &ps)
// Read ch and check for break-in and other awkward values
{ if (feof(data)) { ps = nodata; ch = ' '; return; }
  ch = getc(data);
  if (ch == ESC) ps = finished;
  if (ch < ' ' || feof(data)) ch = ' ';
}

int hexdigit(char ch)
// Convert CH to equivalent value
{ if (ch >= 'a' && ch <= 'e') return(ch + 10 - 'a');
  if (ch >= 'A' && ch <= 'E') return(ch + 10 - 'A');
  if (isdigit(ch)) return(ch - '0');
  else return(0);
}

int getnumber(FILE *data, int base, status &ps)
// Read number in required base
{ bool negative = false;
  char ch;
  int num = 0;
  do
  { readchar(data, ch, ps);
  } while (!(ch > ' ' || feof(data) || ps != running));
  if (ps == running)
  { if (feof(data))
      ps = nodata;
    else
    { if (ch == '-') { negative = true; readchar(data, ch, ps); }
      else if (ch == '+') readchar(data, ch, ps);
      if (!isxdigit(ch))
        ps = baddata;
      else
      { while (isxdigit(ch) && ps == running)
        { if (hexdigit(ch) < base && num <= (maxint - hexdigit(ch)) / base)
            num = base * num + hexdigit(ch);
          else
            ps = baddata;
          readchar(data, ch, ps);
        }
      }
    }
    if (negative) num = -num;
    if (num > 0)
      return num % 256;
    else
      return (256 - abs(num) % 256) % 256;
  }
  return 0;
}

void MC::emulator(MC_bytes initpc, FILE *data, FILE *results, bool tracing)
{ MC_bytes pcnow;           // Old program count
  MC_bytes carry;           // Value of carry bit

  cpu.z = false; cpu.p = false; cpu.c = false; // initialize flags
  cpu.a = 0;     cpu.x = 0;     cpu.sp = 0;    // initialize registers
  cpu.pc = initpc;                             // initialize program counter
  ps = running;
  do
  { cpu.ir = mem[cpu.pc];   // fetch
    pcnow = cpu.pc;         // record for use in tracing/postmortem
    increment(cpu.pc);      // and bump in anticipation
    if (tracing) trace(results, pcnow);
    switch (cpu.ir)         // execute
```

```
{ case MC_nop:
    break;
  case MC_cla:
    cpu.a = 0; break;
  case MC_clc:
    cpu.c = false; break;
  case MC_clx:
    cpu.x = 0; break;
  case MC_cmc:
    cpu.c = !cpu.c; break;
  case MC_inc:
    increment(cpu.a); setflags(cpu.a); break;
  case MC_dec:
    decrement(cpu.a); setflags(cpu.a); break;
  case MC_inx:
    increment(cpu.x); setflags(cpu.x); break;
  case MC_dex:
    decrement(cpu.x); setflags(cpu.x); break;
  case MC_tax:
    cpu.x = cpu.a; break;
  case MC_ini:
    cpu.a = getnumber(data, 10, ps); setflags(cpu.a); break;
  case MC_inb:
    cpu.a = getnumber(data, 2, ps); setflags(cpu.a); break;
  case MC_inh:
    cpu.a = getnumber(data, 16, ps); setflags(cpu.a); break;
  case MC_ina:
    char ascii;
    readchar(data, ascii, ps);
    if (feof(data)) ps = nodata;
    else { cpu.a = ascii; setflags(cpu.a); }
    break;
  case MC_oti:
    if (cpu.a < 128)
      fprintf(results, "%d ", cpu.a);
    else
      fprintf(results, "%d ", cpu.a - 256);
    if (tracing) putc('\n', results);
    break;
  case MC_oth:
    fprintf(results, "%02X ", cpu.a);
    if (tracing) putc('\n', results);
    break;
  case MC_otc:
    fprintf(results, "%d ", cpu.a);
    if (tracing) putc('\n', results);
    break;
  case MC_ota:
    putc(cpu.a, results);
    if (tracing) putc('\n', results);
    break;
  case MC_otb:
    int bits[8];
    MC_bytes number = cpu.a;
    for (int loop = 0; loop <= 7; loop++)
      { bits[loop] = number % 2; number /= 2; }
    for (loop = 7; loop >= 0; loop--)
      fprintf(results, "%d", bits[loop]);
    putc(' ', results);
    if (tracing) putc('\n', results);
    break;
  case MC_psh:
    decrement(cpu.sp); mem[cpu.sp] = cpu.a; break;
  case MC_pop:
    cpu.a = mem[cpu.sp]; increment(cpu.sp); setflags(cpu.a); break;
  case MC_shl:
    cpu.c = (cpu.a * 2 > 255); cpu.a = cpu.a * 2 % 256;
    setflags(cpu.a); break;
  case MC_shr:
    cpu.c = cpu.a & 1; cpu.a /= 2; setflags(cpu.a); break;
  case MC_ret:
    cpu.pc = mem[cpu.sp]; increment(cpu.sp); break;
  case MC_hlt:
    ps = finished; break;
  case MC_lda:
    cpu.a = mem[mem[cpu.pc]]; increment(cpu.pc); setflags(cpu.a); break;
  case MC_ldx:
    cpu.a = mem[index()]; increment(cpu.pc); setflags(cpu.a); break;
  case MC_ldi:
    cpu.a = mem[cpu.pc]; increment(cpu.pc); setflags(cpu.a); break;
  case MC_lsp:
    cpu.sp = mem[mem[cpu.pc]]; increment(cpu.pc); break;
  case MC_lsi:
```

```
  cpu.sp = mem[cpu.pc]; increment(cpu.pc); break;
case MC_sta:
  mem[mem[cpu.pc]] = cpu.a; increment(cpu.pc); break;
case MC_stx:
  mem[index()] = cpu.a; increment(cpu.pc); break;
case MC_add:
  cpu.c = (cpu.a + mem[mem[cpu.pc]] > 255);
  cpu.a = (cpu.a + mem[mem[cpu.pc]]) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_adx:
  cpu.c = (cpu.a + mem[index()] > 255);
  cpu.a = (cpu.a + mem[index()]) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_adi:
  cpu.c = (cpu.a + mem[cpu.pc] > 255);
  cpu.a = (cpu.a + mem[cpu.pc]) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_adc:
  carry = cpu.c;
  cpu.c = (cpu.a + mem[mem[cpu.pc]] + carry > 255);
  cpu.a = (cpu.a + mem[mem[cpu.pc]] + carry) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_acx:
  carry = cpu.c;
  cpu.c = (cpu.a + mem[index()] + carry > 255);
  cpu.a = (cpu.a + mem[index()] + carry) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_aci:
  carry = cpu.c;
  cpu.c = (cpu.a + mem[cpu.pc] + carry > 255);
  cpu.a = (cpu.a + mem[cpu.pc] + carry) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sub:
  cpu.c = (cpu.a < mem[mem[cpu.pc]]);
  cpu.a = (cpu.a - mem[mem[cpu.pc]] + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sbx:
  cpu.c = (cpu.a < mem[index()]);
  cpu.a = (cpu.a - mem[index()] + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sbi:
  cpu.c = (cpu.a < mem[cpu.pc]);
  cpu.a = (cpu.a - mem[cpu.pc] + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sbc:
  carry = cpu.c;
  cpu.c = (cpu.a < mem[mem[cpu.pc]] + carry);
  cpu.a = (cpu.a - mem[mem[cpu.pc]] - carry + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_scx:
  carry = cpu.c;
  cpu.c = (cpu.a < mem[index()] + carry);
  cpu.a = (cpu.a - mem[index()] - carry + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_sci:
  carry = cpu.c;
  cpu.c = (cpu.a < mem[cpu.pc] + carry);
  cpu.a = (cpu.a - mem[cpu.pc] - carry + 256) % 256;
  increment(cpu.pc); setflags(cpu.a); break;
case MC_cmp:
  cpu.c = (cpu.a < mem[mem[cpu.pc]]);
  setflags((cpu.a - mem[mem[cpu.pc]] + 256) % 256);
  increment(cpu.pc); break;
case MC_cpx:
  cpu.c = (cpu.a < mem[index()]);
  setflags((cpu.a - mem[index()] + 256) % 256);
  increment(cpu.pc); break;
case MC_cpi:
  cpu.c = (cpu.a < mem[cpu.pc]);
  setflags((cpu.a - mem[cpu.pc] + 256) % 256);
  increment(cpu.pc); break;
case MC_ana:
  cpu.a &= mem[mem[cpu.pc]];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
case MC_anx:
  cpu.a &= mem[index()];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
case MC_ani:
  cpu.a &= mem[cpu.pc];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
case MC_ora:
  cpu.a |= mem[mem[cpu.pc]];
  increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
```

```
      case MC_orx:
        cpu.a |= mem[index()];
        increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
      case MC_ori:
        cpu.a |= mem[cpu.pc];
        increment(cpu.pc); setflags(cpu.a); cpu.c = false; break;
      case MC_brn:
        cpu.pc = mem[cpu.pc]; break;
      case MC_bze:
        if (cpu.z) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bnz:
        if (!cpu.z) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bpz:
        if (cpu.p) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bng:
        if (!cpu.p) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bcs:
        if (cpu.c) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_bcc:
        if (!cpu.c) cpu.pc = mem[cpu.pc]; else increment(cpu.pc); break;
      case MC_jsr:
        decrement(cpu.sp);
        mem[cpu.sp] = (cpu.pc + 1) % 256; // push return address
        cpu.pc = mem[cpu.pc]; break;
      default:
        ps = badop; break;
    }
  } while (ps == running);
  if (ps != finished) postmortem(results, pcnow);
}

void MC::interpret(void)
{ char filename[256];
  FILE *data, *results;
  bool tracing;
  int entry;
  printf("\nTrace execution (y/N/q)? ");
  char reply = getchar(); scanf("%*[^\n]"); getchar();
  if (toupper(reply) != 'Q')
  { tracing = toupper(reply) == 'Y';
    printf("\nData file [STDIN] ? "); gets(filename);
    if (filename[0] == '\0') data = NULL;
    else data = fopen(filename, "r");
    if (data == NULL)
      { printf("taking data from stdin\n"); data = stdin; }
    printf("\nResults file [STDOUT] ? "); gets(filename);
    if (filename[0] == '\0') results = NULL;
    else results = fopen(filename, "w");
    if (results == NULL)
      { printf("sending results to stdout\n"); results = stdout; }
    printf("Entry point? ");
    if (scanf("%d%*[^\n]", &entry) != 1) entry = 0; getchar();
    emulator(entry % 256, data, results, tracing);
    if (results != stdout) fclose(results);
    if (data != stdin) fclose(data);
  }
}

MC::MC()
{ for (int i = 0; i <= 255; i++) mem[i] = MC_bad;
  // Initialize mnemonic table
  for (i = 0; i <= 255; i++) mnemonics[i] = "???";
  mnemonics[MC_aci] = "ACI";  mnemonics[MC_acx] = "ACX";
  mnemonics[MC_adc] = "ADC";  mnemonics[MC_add] = "ADD";
  mnemonics[MC_adi] = "ADI";  mnemonics[MC_adx] = "ADX";
  mnemonics[MC_ana] = "ANA";  mnemonics[MC_ani] = "ANI";
  mnemonics[MC_anx] = "ANX";  mnemonics[MC_bcc] = "BCC";
  mnemonics[MC_bcs] = "BCS";  mnemonics[MC_bng] = "BNG";
  mnemonics[MC_bnz] = "BNZ";  mnemonics[MC_bpz] = "BPZ";
  mnemonics[MC_brn] = "BRN";  mnemonics[MC_bze] = "BZE";
  mnemonics[MC_cla] = "CLA";  mnemonics[MC_clc] = "CLC";
  mnemonics[MC_clx] = "CLX";  mnemonics[MC_cmc] = "CMC";
  mnemonics[MC_cmp] = "CMP";  mnemonics[MC_cpi] = "CPI";
  mnemonics[MC_cpx] = "CPX";  mnemonics[MC_dec] = "DEC";
  mnemonics[MC_dex] = "DEX";  mnemonics[MC_hlt] = "HLT";
  mnemonics[MC_ina] = "INA";  mnemonics[MC_inb] = "INB";
  mnemonics[MC_inc] = "INC";  mnemonics[MC_inh] = "INH";
  mnemonics[MC_ini] = "INI";  mnemonics[MC_inx] = "INX";
  mnemonics[MC_jsr] = "JSR";  mnemonics[MC_lda] = "LDA";
  mnemonics[MC_ldi] = "LDI";  mnemonics[MC_ldx] = "LDX";
  mnemonics[MC_lsi] = "LSI";  mnemonics[MC_lsp] = "LSP";
  mnemonics[MC_nop] = "NOP";  mnemonics[MC_ora] = "ORA";
  mnemonics[MC_ori] = "ORI";  mnemonics[MC_orx] = "ORX";
```

```
  mnemonics[MC_ota] = "OTA";  mnemonics[MC_otb] = "OTB";
  mnemonics[MC_otc] = "OTC";  mnemonics[MC_oth] = "OTH";
  mnemonics[MC_oti] = "OTI";  mnemonics[MC_pop] = "POP";
  mnemonics[MC_psh] = "PSH";  mnemonics[MC_ret] = "RET";
  mnemonics[MC_sbc] = "SBC";  mnemonics[MC_sbi] = "SBI";
  mnemonics[MC_sbx] = "SBX";  mnemonics[MC_sci] = "SCI";
  mnemonics[MC_scx] = "SCX";  mnemonics[MC_shl] = "SHL";
  mnemonics[MC_shr] = "SHR";  mnemonics[MC_sta] = "STA";
  mnemonics[MC_stx] = "STX";  mnemonics[MC_sub] = "SUB";
  mnemonics[MC_tax] = "TAX";
}
```

BIBLIOGRAPHY

Addyman, A.M., Brewer,R., Burnett Hall, D.G. De Morgan, R.M., Findlay, W., Jackson M.I., Joslin, D.A., Rees, M.J., Watt, D.A., Welsh, J. and Wichmann, B.A. (1979) A draft description of Pascal, *Software - Practice and Experience*, **9**(5), 381-424.

Aho, A.V., Sethi, R. and Ullman, J.D. (1986) *Compilers: Principles, Techniques and Tools*, Addison- Wesley, Reading, MA.

Alblas, H. and Nymeyer, A. (1996) *Practice and Principles of Compiler Building with C*, Prentice-Hall, Hemel Hempstead, England.

Andrews, G.R. and Schneider, F.B. (1983) Concepts and notation for concurrent programming, *ACM Computing Surveys, **15**, 3-43.*

Backhouse, R.C. (1979) *Syntax of Programming Languages: Theory and Practice*, Prentice-Hall, Hemel Hempstead, England.

Bailes, P.A. (1984) A rational Pascal, *Australian Computer Journal*, **16**(4), 155-176.

Barron, D.W. (ed) (1981) *Pascal - the Language and its Implementation*, Wiley, Chichester, England.

Ben-Ari, M. (1982) *Principles of Concurrent Programming*, Prentice-Hall, Englewood Cliffs, NJ.

Bennett, J.P. (1990) *Introduction to Compiling Techniques: a First Course using ANSI C, LEX and YACC*, McGraw-Hill, London.

Bjorner, D. and Jones, C.B. (eds) (1982) *Formal Specification and Software Development*, Prentice-Hall, Hemel Hempstead, England.

Bratman, H. (1961) An alternate form of the Uncol diagram, *Communications of the ACM*, **4**(3), 142.

Brinch Hansen, P. (1983) *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1985) *On Pascal Compilers*, Prentice-Hall, Englewood Cliffs, NJ.

Burns, A. and Davies, G. (1993) *Concurrent Programming*, Addison-Wesley, Wokingham, England.

Burns, A. and Welling, A. (1989) *Real Time Systems and their Programming Languages*, Addison-Wesley, Wokingham, England.

Bustard, D.W., Elder, J. and Welsh, J. (1988) *Concurrent Programming Structures*, Prentice-Hall, Hemel Hempstead, England.

Cailliau, R. (1982) How to avoid getting schlonked by Pascal, *ACM SIGPLAN Notices*, **17**(12),

31-40.

Chomsky, N. (1959) On certain formal properties of grammars, *Information and Control*, **2**(2), 137-167.

Cichelli, R.J. (1979) A class of easily computed, machine independent, minimal perfect hash functions for static sets, *Pascal News* **15**, 56-59.

Cichelli, R.J. (1980) Minimal perfect hash functions made simple, *Communications of the ACM*, **23**(1), 17- 19.

Cooper, D. (1983) *Standard Pascal Reference Manual*, Norton, New York.

Cormack, G.V., Horspool, R.N.S. and Kaiserwerth, M. (1985) Practical perfect hashing, *Computer Journal* **28**(1), 54-58.

Cornelius. B.J., Lowman. I.R. and Robson, D.J. (1984) Steady-state compilers, *Software - Practice and Experience*, **14**(8), 705-709.

Cornelius, B.J. (1988) Problems with the language Modula-2, *Software - Practice and Experience*, **18**(6), 529- 543.

Dobler, H. and Pirklbauer, K. (1990) Coco-2 - a new compiler-compiler, *ACM SIGPLAN Notices*, **25**(5), 82- 90.

Dobler, H. (1991) Top-down parsing in Coco-2, *ACM SIGPLAN Notices*, **26**(3), 79-87.

Earley, J. and Sturgis, H. (1970) A formalism for translator interactions, *Communications of the ACM*, **13**(10), 607-617.

Elder, J. (1994) *Compiler Construction: a recursive descent model*, Prentice-Hall, Hemel Hempstead, England.

Ellis, M.A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA.

Fischer, C.N. and LeBlanc, R.J. (1988) *Crafting a Compiler*, Benjamin Cummings, Menlo Park, CA.

Fischer, C.N. and LeBlanc, R.J. (1991) *Crafting a Compiler with C*, Benjamin Cummings, Menlo Park, CA.

Gough, K.J. (1988) *Syntax Analysis and Software Tools*, Addison-Wesley, Wokingham, England.

Gough, K.J. and Mohay, G.M. (1988) *Modula-2: A Second Course in Programming*, Prentice-Hall, Sydney, Australia.

Grosch, J. (1988) Generators for high-speed front ends, *Lecture Notes in Computer Science*, 371, 81-92, Springer, Berlin.

Grosch, J. (1989) Efficient generation of lexical analysers, *Software - Practice and Experience*,

19(11), 1089- 1103.

Grosch, J. (1990a) Lalr - a generator for efficient parsers, *Software - Practice and Experience*, **20**(11), 1115- 1135.

Grosch, J. (1990b) Efficient and comfortable error recovery in recursive descent parsers, *Structured Programming*, **11**, 129-140.

Grune, D. and Jacobs, C.J.H. (1988) A programmer-friendly LL(1) parser generator, *Software - Practice and Experience*, **18**(1), 29-38.

Hennessy, J.L. and Mendelsohn, N. (1982) Compilation of the Pascal case statement, *Software - Practice and Experience*, **12**(9), 879-882.

Hennessy, J.L. and Patterson, D.A. (1990) *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA.

Hoare, C.A.R. and Wirth, N. (1973) An axiomatic definition of the programming language Pascal, *Acta Informatica*, **2**, 335-355.

Holmes, J. (1995) *Object-Oriented Compiler Construction*, Prentice-Hall, Englewood Cliffs, NJ.

Holub, A.I. (1990) *Compiler Design in C*, Prentice-Hall, Englewood Cliffs, NJ.

Hunter, R.B. (1981) *The Design and Construction of Compilers*, Wiley, New York.

Hunter, R.B. (1985) *The Design and Construction of Compilers with Pascal*, Wiley, New York.

Johnson, S.C. (1975) Yacc - Yet Another Compiler Compiler, *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill, NJ.

Kernighan, B.W. (1981) Why Pascal is not my favorite programming language, *Computer Science Technical Report 100*, AT&T Bell Laboratories, Murray Hill, NJ.

Kernighan, B.W. and Ritchie, D.M. (1988) *The C Programming Language* (2nd edn), Prentice-Hall, Englewood Cliffs, NJ.

King, K.N. (1996) *C Programming - a modern approach*, W.W. Norton, New York.

Lecarme, O. and Peyrolle-Thomas, M.C. (1973) Self compiling compilers: an appraisal of their implementations and portability, *Software - Practice and Experience*, 8(2), 149-170.

Lee, J.A.N. (1972) The formal definition of the BASIC language, *Computer Journal*, **15**, 37-41.

Lee, J.A.N. and Sammet, J.E. (1978) History of Programming Languages Conference HOPL I, *ACM SIGPLAN Notices*, **13**(8).

Lee, J.A.N. and Sammet, J.E. (1993) History of Programming Languages Conference HOPL II, *ACM SIGPLAN Notices*, **28**(3).

Lesk, M.E. (1975) Lex - a lexical analyser generator, *Computing Science Technical Report* 39,

AT&T Bell Laboratories, Murray Hill, NJ.

Levine, J.R., Mason, T. and Brown D. (1992) *Lex and Yacc* (2nd edn), O'Reilly and Associates, Sebastapol, CA.

MacCabe, A.B. (1993) *Computer Systems: Architecture, Organization and Programming*, Irwin, Boston, MA.

Mak, R. (1991) *Writing Compilers and Interpreters: an Applied Approach*, John Wiley, New York.

McGettrick, A.D. (1980) *The Definition of Programming Languages*, Cambridge University Press, Cambridge, England.

Meek, B.M. (1990) The static semantics file, *ACM SIGPLAN Notices*, **25**(4), 33-42.

Mody, R.P. (1991) C in education and software engineering, *ACM SIGCSE Bulletin*, **23**(3), 45-56.

Mössenböck, H. (1986) Alex: a simple and efficient scanner generator, *ACM SIGPLAN Notices*, **21**(12), 139- 148.

Mössenböck, H. (1990a) *Coco/R: A generator for fast compiler front ends*, Report 127, Departement Informatik, Eidgenössische Technische Hochschule, Zürich.

Mössenböck, H. (1990b) *A generator for production quality compilers*, in Proceedings of the Third International Workshop on Compiler-Compilers, Lecture Notes in Computer Science 471, Springer, Berlin.

Naur, P. (1960) Report on the algorithmic language Algol 60, *Communications of the ACM*, **3**, 299-314.

Naur, P. (1963) Revised report on the algorithmic language Algol 60, *Communications of the ACM*, **6**(1), 1- 17.

Nori, K.V., Ammann, U., Jensen, K. *et al.* (1981) Pascal-P implementation notes, in *Pascal - the Language and its Implementation*, Barron, D.W. (ed) Wiley, Chichester, England.

Panti, M. and Valenti, S. (1992) A modulus oriented hash function for the construction of minimal perfect tables, *ACM SIGPLAN Notices*, **27**(11), 33-38.

Parr, T.J., Dietz, H.G. and Cohen W.E. (1992) PCCTS 1.00: The Purdue Compiler Construction Tool Set, *ACM SIGPLAN Notices*, **27**(2), 88-165.

Parr, T.J. and Quong, R.W. (1995) ANTLR: A predicated-LL(k) parser generator, *Software - Practice and Experience*, **25**(7), 789-810.

Parr, T.J. and Quong, R.W. (1996) LL and LR Translators need k > 1 lookahead, *ACM SIGPLAN Notices*, **31**(2), 27-34.

Parr, T.J. (1996) *Language translation using PCCTS and C++ (a Reference Guide)*, Automata Publishing, San Jose, CA.

Pemberton, S. (1980) Comments on an error-recovery scheme by Hartmann, *Software - Practice and Experience*, **10**(3), 231-240.

Pemberton, S. and Daniels, M. (1982) *Pascal Implementation - the P4 Compiler*, Ellis Horwood, Chichester.

Pittman, T. and Peters, J. (1992) *The Art of Compiler Design*, Prentice-Hall, Englewood Cliffs, NJ.

Rechenberg, P. and Mössenböck, H. (1989) *A Compiler Generator for Microcomputers*, Prentice-Hall, Hemel Hempstead, England.

Rees, M. and Robson, D. (1987) *Practical Compiling with Pascal-S*, Addison-Wesley, Wokingham, England.

Sakkinen, M. (1992) The darker side of C++ revisited, *Structured Programming*, **13**(4), 155-178.

Sale, A.H.J. (1979) A note on scope, one-pass compilers, and Pascal, *Australian Computer Science Communications*, **1**(1), 80-82. Reprinted in *Pascal News*, **15**, 62-63.

Sale, A.H.J. (1981) The implementation of case statements in Pascal, *Software - Practice and Experience*, **11**(9), 929-942.

Schreiner, A.T. and Friedman, H.G. (1985) *Introduction to Compiler Construction with UNIX*, Prentice-Hall, Englewood Cliffs, NJ.

Sebesta, R.W. and Taylor, M.A. (1985) Minimal perfect hash functions for reserved word lists, *ACM SIGPLAN Notices*, **20**(12), 47-53.

Stirling, C. (1985) Follow set error recovery, *Software - Practice and Experience*, **15**(8), 239-257.

Stroustrup, B. (1990) *The C++ Programming Language* (2nd edn), Addison-Wesley, Reading, MA.

Stroustrup, B. (1993) *The Design and Evolution of C++*, Addison-Wesley, Reading, MA.

Terry, P.D. (1986) *Programming Language Translation*, Addison-Wesley, Wokingham, England.

Terry, P.D. (1995) Umbriel: another minimal programming language, *ACM SIGPLAN Notices*, **30**(5), 11- 17.

Topor, R.W. (1982) A note on error recovery in recursive descent parsers, *ACM SIGPLAN Notices*, **17**(2), 37- 40.

Tremblay, J.P. and Sorenson, P.G. (1985) *Theory and Practice of Compiler Writing*, McGraw-Hill, New York.

Trono, J.A. (1995) A comparison of three strategies for computing letter-oriented, minimal perfect hashing functions, *ACM SIGPLAN Notices*, **30**(4), 29-35.

Ullmann, J.R. (1994) *Compiling in Modula-2 - a First Introduction to Classical Recursive Descent Compiling*, Prentice-Hall, Hemel Hempstead, England.

van den Bosch, P.N. (1992) A bibliography on syntax error handling in context free languages, *ACM SIGPLAN Notices*, **27**(4), 77-86.

Waite, W.M. and Goos, G. (1984) *Compiler Construction*, Springer, New York.

Wakerly, J.F. (1981) *Microcomputer Architecture and Programming*, Wiley, New York.

Watson, D. (1989) *High-level Languages and their Compilers*, Addison-Wesley, Wokingham, England.

Watt, D.A. (1991) *Programming Language Syntax and Semantics*, Prentice-Hall, Hemel Hempstead, England.

Watt, D.A. (1993) *Programming Language Processors*, Prentice-Hall, Hemel Hempstead, England.

Welsh, J. and Hay, A. (1986) *A Model Implementation of Standard Pascal*, Prentice-Hall, Hemel Hempstead, England.

Welsh, J. and McKeag, M. (1980) *Structured System Programming*, Prentice-Hall, Hemel Hempstead, England.

Welsh, J. and Quinn, C. (1972) A Pascal compiler for ICL 1900 series computers, *Software - Practice and Experience*, **2**(1), 73-78.

Welsh, J., Sneeringer, W.J. and Hoare, C.A.R. (1977) Ambiguities and Insecurities in Pascal, *Software - Practice and Experience*, **7**(6), 685-696. (Reprinted in Barron (1981))

Wilson, I.P. and Addyman, A.M. (1982) *A Practical Introduction to Pascal - with BS6192*, MacMillan, London.

Wirth, N. (1974) *On the design of programming languages*, Proc IFIP Congress 74, 386-393, North-Holland, Amsterdam.

Wirth, N. (1976a) *Programming languages - what to demand and how to assess them*, Proc. Symposium on Software Engineering, Queen's University, Belfast.

Wirth, N. (1976b) *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1977) What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, **20**(11), 822-823.

Wirth, N. (1981) Pascal-S: A subset and its implementation, in *Pascal - the Language and its Implementation*, Barron, D.W. (ed), Wiley, Chichester, England.

Wirth, N. (1985) *Programming in Modula-2* (3rd edn), Springer, Berlin.

Wirth, N. (1986) *Compilerbau*, Teubner, Stuttgart.

Wirth, N. (1988a) From Modula to Oberon, *Software - Practice and Experience*, **18**(7), 661-670.

Wirth, N. (1988b) The programming language Oberon, *Software - Practice and Experience*, **18**(7),

671-690.

Wirth, N. (1996) *Compiler Construction*, Addison-Wesley, Wokingham, England.

INDEX

The page numbers relate to a version of the text typeset for A4 paper which is down-loadable from http://scifac.ru.ac.za/compilers/longpcl.zip as a set of PCL files suitable for direct printing to a LaserJet® compatible printer.

8-bit microprocessor 229

Accumulator 25, 30
Absolute addressing 28, 46
Abstract machine 7
Abstract syntax tree 10, 101, 181, 232, 270
Accumulator machine 27
Activation record 246, 263
Actual parameter 93, 259, 263
Adding machine 162
Address 71
    absolute 28
    field expressions 87
    fields 72
    immediate 28
    indexed 29
    indirect 29
    mode 27
    relative 29, 46
    return 246
    run-time 247, 250, 252, 267
Alex scanner generator 176
Alias 259
Algol 60 report 49, 54, 106
Alphabet 50
ALU 30, 39
Ambiguity 49, 133
Ambiguous grammar 104, 125, 133
Analytic phase 8
Anonymous type 215
Antecedent 70
ANTLR parser generator 148
Applied occurrence 71, 191, 208
Argument 259
Arithmetic expression 100, 151, 190
Arithmetic logic unit 30, 39
ARM (Annotated Reference Manual) 70
Array handling 212, 216, 219, 228, 268, 275
ASCII 197, 202
ASSEMBLER
    grammar 72
    language 14, 20, 71
Assembly 7
    assembler class 75
    code generation 75, 82
    conditional 95
    lexical analysis 77
    macro processing 92
    minimal 37
    one-pass 74, 83
    semantic analysis 81
    source handling 76
    syntax analysis 79
    two-pass 73, 80
AST 10, 101, 181, 232, 270
AST code generation 181, 232, 270
Asynchronous input 297
Attribute grammar 69, 110, 151, 157
Augmented grammar 123
    driver program 174
    execution 162
    frame file 147, 161, 175
    ftp sites Appendix A
    installation 161, Appendix A
    makefile 162
    parser interface 173
    scanner interface 166
    support modules 172
Cocol 61, 72, 161

Automata theory 110, 143
Axiomatic semantics 69

Back end 8, 182, 217
Backpatching 74, 219, 224, 262
Backtracking 108, 117
Backus-Naur form 49, 54
Bailes 205
Base class 183, 201, 213, 232, 270
Base pointer 39, 48, 247
BASIC PRINT statement 109
Beacon symbol 137, 211
Binary search 201, 214
Binary tree 82, 214, 244
Block structure 241
BNF 49, 54
    notation 54
    extensions 59
Boolean operator 12, 46, 65, 180, 190
Bootstrap 2, 20, 22, 62
Bottom-up parsing 143
Bound checks 46, 228
Bounded buffer 286
break statement 204
Brinch Hansen 205, 216, 251, 263, 275
British Standard for EBNF 61
Bus lines 25
Busy-wait 295

C declarations 177
Call-by-reference 259, 274
Call-by-value 259, 274
Canonical derivation 56, 104
CASE statement 204, 225
Character handler 8
Character set 63, 164
Chomsky hierarchy 107
Cichelli 202
Clang
    Cocol specification 110, 220, Appendix C
    code generator interface 212, 217
    constraint analyser 208
    error handling 198, 206
    hand-crafted parser 203, 206
    hand-crafted scanner 199
    hand-crafted source handler 196
    report (on Diskette)
    syntax 111
Clang-Topsy translator 192
Closure 50
    Kleene 50, 59, 65
    positive 50
COBEGIN ... COEND 282, 293
Cocktail 147
Coco-2 176
Coco/R 61, 123, 146, 161, 189
    Clang error handling 208
    Clang scanner 201
    Clang specification 110, 220, Appendix C
Concrete syntax 10
Concrete syntax tree 10
Concurrent processes 281
Concurrent programming 281
Condition flag 30, 36, 46
Conditional assembly 95
Consequence rule 69
Consequent 70
Constant declarations 124
Constant expressions 236

actions 168
ANY 164, 168
CHARACTERS 164
COMMENTS 164
CONTEXT clause 165
EOF 170
formal attributes 167
grammar checks 171
IGNORE 165
LexString 173
NAMES 166
parser specification 167
Pragmas 162, 166
PRAGMAS 166
PRODUCTIONS 167
scanner interface 166
scanner specification 163
semantic errors 172
SemError() 173
Successful() 173
SynError() 173
SYNC 168, 170, 208
synchronization sets 170, 207
syntax error recovery 162, 169
TOKENS 165
user names 166
WEAK 168, 208
weak separators 171
weak terminal 168, 170
Code generation 12
ASSEMBLER 82
assembler code 238
from an AST 181, 232, 270
generator interface 212, 217
native 231
on-the-fly 179, 184, 226, 250, 255, 266
one-address code 179, 181
optimization 12, 181, 235
reentrant code 246
stack machine code 226
Collision 91
Comments 51, 71, 109, 164
pragmatic 201
Communication 284
Compile-time 2, 71
Compiler 2, 7
load-and-go 7, 37, 219
multi-pass 14
one-pass 14
structure 8, 195
Compiler generator 4, 146
Dynamic link 247, 253
Dynamic semantics 4, 68

ε-free grammar 103
ε-productions 58, 107, 109
EBNF 59
British standard 61
expressions 60
Edison 70, 216, 225, 284
Effective address 28, 30, 39
Electronic mail 193
Empty set 51
Empty statement 128
Empty string 51, 186
Emulation 25
Emulator 16
single-accumulator 35, Appendix D
stack machine 42, Appendix B
Environment 156
Equivalent grammar 100, 125, 133
Erasure 59, 107
Error checking code 13, 275
Error
context free 136, 169, 206
context-sensitive 172, 208
correction 13, 80, 136
detection 80, 86, 136
handling 13
recovery 13, 136, 206
reporting 13, 87, 198
semantic 172
spurious 136, 138
Constant folding 184, 235
Constraint analyser 10, 73, 208
CONTEXT clause 165
Context condition 157
Context-free grammar 109, 135
Context-sensitive 73, 106, 262
Context switch 291
Contextual constraint analyser 10
continue statement 204
Control statement 219
Conway's problem 286
Critical region 284, 288
Cross compiler 2, 7, 17, 22, 75
Cross reference generator 191
Cycle-free grammar 104

DAG (directed acyclic graph) 237
Dangling ELSE 105, 126, 133, 204, 223
Data register 25
Deadlock 285, 295
Debugger 239
Declaration level 243
Declaration order 278
Declarations 81, 138, 208, 241, 245, 260, 262
Decompiler 7
Decorated tree 10, 152
Defining occurrence 71, 208
Delphi 3
Denotational semantics 69
Dereferencing 41
Derived from 54
Descriptor ring 291
Designator 134, 209, 212, 252, 259, 273
Deterministic finite automaton (DFA) 142
Deterministic parsing 116
DFA (deterministic finite automaton) 142
Dijkstra 284
Direct addressing 28
Directed acyclic graph (DAG) 237
Directive
assembler 71
compiler 201
table 74
Directly produces 54
Director set 123
Disassembler 7
Display 252, 257
Display copy 253
Distributed processing 290
DLG 148
Driver program 174, 196
Free Software Foundation 19, 202
Front end 8, 182, 195
FSA (Finite state automata) 110, 139, 201
Function 259
assignment 264
reference 260
return value 265, 277

GNU project 19, 202
Goal symbol 53, 55
Goods trains 65
GOTO statement 225, 258
Gough 202
Global variables 163, 242, 292
gperf 202
Grammar 53
ambiguous 104, 125, 133
attribute 69, 110, 151, 157
augmented 123
checks 171
context-free 109, 135
context-sensitive 106
cycle free 104
ε-free 103
equivalent 100, 125, 133
expression 57, 100, 149, 151, 179, 190
graph representation 185
hierarchy 107
L-attributed 157
left-linear 110
null free 103
reduced 103

Escape sequence 200
EXIT statement 204, 224
Exception 13
Exclusion 284
Expression
    evaluation 151, 190
    grammar 57, 100, 149, 151, 179, 190
    parameter 259
Extended BNF 59
Extent of identifiers 242

Fetch-execute cycle 26, 35, 43
Finite state automata (FSA) 110, 139, 162
FIRST function 118, 129, 136
FOLLOW function 120, 129, 136, 206, 211
Followers 137, 206
FOR loop 204, 223, 279
Formal
    attributes 168
    language 50
    methods 4
    parameter 93, 259
    semantics 67
Fortran 2, 9, 50, 60, 117, 241
Forward
    declaration 257
    reference 73, 83, 85, 88, 159, 219, 238, 251
Frame file 147, 161, 175
Frame header 246, 263
Indexed addressing 29
Indivisible operation 284, 295
Inductive expression 69
Inference rule 69
Inherent addressing 28
Inherited attribute 157
Instruction pointer 26
Instruction register 25
Instruction set 31, 40
Intermediate addressing 247
Intermediate code 11
Interpreter 15
Interpretive compiler 15, 22
Interrupts 26
IOTRANSFER 297

Java 4

Kernel 290
Keywords 10, 64, 125, 142, 200, 201
Kleene closure 50, 59, 65

L-attributed grammar 157
L-value 212
Label 71, 81, 99
    redefined 82, 85
    undefined 82
LALR parsing 146
Lambda production 59
Language 50
Language design 5, 276, 280
Left canonical derivation 56
Left linear grammar 110
Left recursion 54, 126, 152
Level, declaration 243
lex scanner generator 147
Lexeme 142, 163, 199, 201
Lexical analyser 8, 58, 77, 89
Lexical structure 58, 61, 63
Lexicon 58
LexName 173
LexString 173
Lifetime of identifiers 242
Link
    dynamic 247, 253
    static 248, 257
Linkage area 246
Linkage editor 7
Linker 7
Literal pool 39, 48
Literal strings 115, 199, 228
LL(1) conflict resolution 210, 223, 241, 245, 266, 269
LL(1) restrictions 118, 125
LL(k) parsing 117, 146
    regular 109
    restrictions 102, 118, 125
    right-linear 109
    S-attributed 157
    transformation 125
    type 0 107
    type 1 107
    type 2 109
    type 3 109
Graph 185

Half bootstrap 21
Hand-crafted parser 203, 206
Hand-crafted scanner 139, 199
Hand-crafted source handler 196
Hash table 90, 214, 245
Hashing function 90
Hexadecimal literals 78
High-level translator 7, 14, 19, 192
Highland Gathering 66
Host language 2, 19
Hypothetical stack machine 38, 217, 226

IF ... THEN ... ELSE 70, 105, 125, 133, 204, 2
Ignorable characters 63, 164
Immediate addressing 28
Implementation language 2
Independent compilation 14
Index register 30
Loop
    EXIT 224
    FOR 204, 223, 279
    REPEAT 125, 223
    WHILE 10, 68, 70, 205, 219
LR(k) parsing 143

Machine
    abstract 7
    code 231
    dependencies 8
    instructions 25
    single-accumulator 30
    stack 38
Macro 92
    assembler 7, 74, 92
    handler class 93
    expansion 92, 92
    parameters 93
    recursive 96
Make file 66, 162
Mark stack pointer 39, 248, 263
Memory-indexed addressing 29
Memory-indirect addressing 29
Metalanguage 49
Metasymbol 50, 59
Minimal perfect hashing function 202
Mnemonic 26, 71
Modularity 4
Multi-stage translator 14
Music 193
Mutual exclusion 284
Mutually recursive procedures 257

Native code 231
Name equivalence 215
NDFA (non-deterministic finite automaton) 142
Nested blocks 242
Non-deterministic finite automaton (NDFA) 142
Non-reachable production 103
Non-terminal 53, 60
Non-terminating production 103
Null
    production 58
    string 50
    string problem 120
Nullable 59, 119, 130, 134

Oberon 2, 3, 216, 240, 278
Object language 2
Object orientation 3, 132, 183, 235
occam 284
Offset 226, 247, 263, 289
On-the-fly code generator 226

LLGen parser generator 147
Load-and-go translator 7, 219
Loader 37
Local variables 220, 245
Location counter 75, 212
Logical operator 180
LOOP ... EXIT ... END statements 204, 224
Optimization 12, 181, 235, 251
    Peephole 12, 228, 231, 257
Orthogonality 4
Overflow 36, 46, 185

P-code assembler 23
P-machine 17
Parameters
    actual 93, 259, 263, 267, 271
    array 268, 275
    formal 259, 262
    passing mechanisms 259, 265, 268
    procedural 275
Parameterless procedure 241
Parse stack 144
Parse tree 101, 152
Parser 10, 58
    bottom-up 143
    construction 129
    deterministic 116
    generator 146, 185
    interface 173
    LALR 146
    LL(k) 117
    LR(k) 143
    recursive descent 116, 129, 132, 149, 195
    SLR 146
    specification 167
    table driven 141, 145
    top-down 116, 143
Pascal standard 70
Pascal-FC 284
Pascal-P compiler 17, 22, 203, 225
Pascal-Plus 284
Pascal-S 239, 257, 298
Passes 14, 195
PCCTS compiler generator 148
Peephole optimization 12, 228, 231, 257
Perfect hashing function 202
Pervasive identifier 244, 275
Phases 8, 195
Phrase 50
Phrase structure 56, 58, 61, 63
Phrase structure tree 56, 101, 151
Pointer types 216
Portability 3, 15, 20
Portable interpretive compilers 17, 22
Porting a compiler 20
Post-mortem dump 275
Postfix notation 150
Pragma 162, 165, 201
Pragmatic considerations 50, 73
Precedence 51, 127, 180
Precedence graph 282
Predeclared identifier 275
Preprocessor 7
Pretty printer 192
Priority queue 297
Procedural parameter 275
Procedure
    activation 248, 263
    grammars 102, 118
    LL(1) 118, 125
Return address 246, 249, 253
RETURN statement 260, 265, 268, 277
Return value 265
Reverse Polish 65, 150
Rewrite rule 54
Right canonical derivation 56
Right linear grammar 109
Right recursion 54, 126
ROM BIOS 33
Round-robin scheduler 296
Rule of inference 69
Rule of consequence 69
Run-time 2, 71

One-address code 27
One and a half address code 27
One-pass assembler 74, 83
Opcode 27, 71
Open array 260, 275
Operating system 2
Operational semantics 33, 68
    calling 249, 252, 255, 266
    declaration 245
    nested 242
    parameters 259
    regular 241
    return 246, 250, 253
Process
    concurrent 281
    descriptor 291
    parallel 282
    priority 297
    sequential 281
Producer-consumer problem 285
Produces 54
Produces directly 54
Production rule 53
Productions 53, 62
    Cocol 169
    null 58, 103
    single 104, 129
    unreachable 107
    useless 103, 128
Program counter 26, 30, 39
Program proving 69
Pseudo-code 16
Push-down automata 143

Qualified identifier 135

R-value 212
Range checks 229
Rational Pascal 205
Real-time 282
Record types 216
Recursion
    in BNF 57, 59
    in grammars 54
    in procedures 254, 257, 276
    left 54, 126, 152
    right 54, 126
Recursive descent parsing 116, 129, 132, 149,
Reduce action 144
Reduced grammar 103
Reductions 143
Redundant code 236
Register 25
    allocation 180
    indexed addressing 29
    indirect addressing 29
    status 30
Regular
    expression 50, 139
    grammar 109
    procedure 168, 241
Rehashing 91
Relative addressing 29, 46
Relocatable code 7, 97
Removal of redundant code 236
REPEAT loop 125, 204, 223
Reserved keywords 10, 64, 125, 142, 200
Restrictions
Side-effect 70, 84, 279
Signal 284, 287, 295
Single-accumulator machine 30
    assembler 37
    emulator 35
    opcodes 31
Single pass compilation 14
Single production 104, 129
SKIP statement 205
Source handling 76, 196, 198
Source language 2
Spreadsheet 15, 190
Spurious error 136, 138
Stack frame 39, 246, 253, 263
Stack machine 38, 217, 226, 230

S-attributed grammar 157
SLR parsing 146
SYNC 168, 170, 208
Sale’s algorithm 277
Scanner 8, 52, 58, 77, 89, 129, 139
    generator 146
    interface 166
    specification 163
Scope
    insecurities 276
    node 243
    rules 242
Self-compiling compiler 20
Self-embedding 53, 54, 143
Self-resident translator 7
Semantic
    action 149, 151
    attributes 152
    driven parser 169, 210
    error detection 172
    overtones 61, 133
Semantics 49
    axiomatic 69
    denotational 69
    dynamic 4, 68
    formal 67
    operational 68
    short-circuit 12, 180, 228, 239
    static 4, 68, 81, 209
Semaphore 284, 287, 291, 295
SemError() 173
Semicolon 128, 212
Sentence 50, 53
Sentential form 54
Sentinel node 243
Separate compilation 7, 14
Sequential algorithms 281
Sequential conjunction 12
Sequential process
Set class 87, 137
Shakespeare 124
Shared memory 290
Shift action 144
Shift-reduce conflict 146
Short-circuit semantics 12, 14, 180, 228, 239
Terminal 53, 60
    start sets and symbols 118
    successors 120
Three-address code 27
Time-slicing 290, 297
Token 8, 50, 165
    classes 163
Tonic Solfa 193
Top-down parsing 116, 143
Topsy 114, 195
    report (on Diskette)
Transition diagram 141
Transmitted attribute 157
Transputer 290
Tree-building actions 181
Turbo Pascal 3, 7, 16, 35, 183, 198, 257
Two-address code 27
Two-pass assembler 73, 80
Type 0 grammar 107
Type 1 grammar 103
Type 2 grammar 109
Type 3 grammar 109
Type checking 10, 214
Type identifiers 205

UCSD Pascal 17, 23, 99
Umbriel 239, 278
Undeclared identifier 82, 214
Union 146, 183, 213, 216, 232
UNIX 2, 66, 151
Unrestricted grammar 107
Useless production 103, 128
User names 166

VDL 69
VDM 69
Value Designator 212, 232
    assembler 47
    emulator 42
    opcodes 40
Stack pointer 30, 39, 48, 230
Start sets 118
Start symbol 53, 55
State diagram 290
State variable 91
Static
    level 243, 247
    link 248
    semantic analyser 10, 81, 87, 208, 266
    semantics 4, 68, 209, 266
Status register 30
Storage management 246
String 50, 199, 228
    table 74, 79
    type 216
Structural equivalence 215
Subrange 25, 200, 216
Subroutine 93
Subset language 20
Successful() 173
switch statement 204
Symbol 50
    beacon 137, 211
    goal 53
Symbol table 13, 73, 80, 89, 135, 209, 212, 2
Synchronization 136, 170, 206, 284
SynError() 173
Syntactic class 53
Syntax 4
    analyser 10, 79, 87
    diagram 67
    equation 54
    error recovery 136, 162
Syntax directed translation 149
Synthesized attribute 154
Synthetic phase 8
Systems program 2

T-diagram 6, 19
Table-driven algorithm 141, 145
Target language 2

Variable
    declarations 138
    designator 212, 259
    offset 226, 247, 263, 289
    redeclared 242
    undeclared 82, 214
Variable Designator 212, 232, 259
Variant record 146, 183, 213, 216, 232
Virtual machine 7, 16, 230
Visibility 242
Vocabulary 53
Void function 129, 168, 241, 244, 257

Wait 284, 287, 295
Weak separator 171, 206
Weak terminal 168, 170
WHILE loop 10, 68, 70, 205, 219
Wirth 3, 59, 189, 206, 240, 257, 277, 298
WITH statement 68

Yacc parser generator 146, 147, 151

Z80 processor 229
Zero-address instruction 40